Von der Carl-Friedrich-Gauß-Fakultät
für Mathematik und Informatik
der Technischen Universität Braunschweig
genehmigte Dissertation
zur Erlangung des Grades eines
Doktor-Ingenieurs (Dr.-Ing.)

Karsten Diethers

Werkzeuggestützte formale Analyse von Echtzeitsystemen

Datum der Promotion: 05.07.06
1. Referentin: Prof. Dr. Ursula Goltz
2. Referent: Prof. Dr. Gregor Engels
eingereicht am: 31.01.06

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

München, den 25.01.2006

Danksagung

Diese Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter an der Technischen Universität Braunschweig. Viele wertvolle Anregungen habe ich dabei aus meiner Projektarbeit im Sonderforschungsbereich 562 bekommen. Meiner Mentorin Prof. Ursula Goltz danke ich für die Ermöglichung meiner Arbeit als wissenschaftlicher Mitarbeiter und für den nötigen wissenschaftlichen Freiraum. Ich bedanke mich, dass mir neben internen Diskussionen zahlreiche Gelegenheiten zum internationalen Erfahrungsaustausch gegeben wurden. Meinen Braunschweiger Kollegen des Instituts für Programmierung und Reaktive Systeme danke ich für die gemeinsame Arbeit und die kollegiale Atmosphäre. Mein besonderer Dank gilt Michaela Huhn für die zahlreichen Diskussionen und Ratschläge. Besonders genossen habe ich die Tätigkeit im SFB 562, in dem fachübergreifendes Arbeiten Herausforderung und Ansporn zugleich war. Stellvertretend für die große Zahl von kompetenten Kollegen in diesem Bereich, die sich hier nicht alle namentlich erwähnen lassen, möchte ich Prof. Dr. Friedrich Wahl als Sprecher des Sonderforschungsbereichs danken.

Meinen Eltern danke ich für die Unterstützung und die Ermöglichung meines Hochschulstudiums. Ohne sie wäre diese Arbeit niemals möglich gewesen. Meinen Freunden, insbesondere Zhen, danke ich für die moralische Unterstützung und Geduld während der Endphase dieser Arbeit.

München, im Dezember 2005
Karsten Diethers

Zusammenfassung

In dieser Arbeit werden Verfahren zur effizienten, benutzerfreundlichen Analyse von Echtzeitsystemen entwickelt. Ziel ist die Verbesserung der Entwurfsqualität hinsichtlich von dynamischen/zeitlichen Programmabläufen möglichst ohne zusätzlichen Aufwand seitens des Entwicklers. Dieses erfordert ein Aufsetzen auf Spezifikationen, die bei der Entwicklung ohnehin anfallen. Konkret wird daher untersucht, wie sich Modelle der Unified Modeling Language mit formalen Methoden analysieren lassen und wie diese Analyse automatisiert werden kann. Es wird geklärt, welche Teilmenge von Modellen als Ausgangspunkt für eine dynamische Analyse geeignet ist. Dabei werden in dieser Arbeit drei Analyseziele definiert, die jeweils eine eigene Sprachdefinition erfordern.

Wichtiger Bestandteil der Arbeit ist die Realisierung einer automatisierten Analyse. Dabei wird auf formale Techniken wie Model-Checking und auf algorithmische Lösungen der Scheduling-Theorie zurückgegriffen. Es wird nachgewiesen, dass sich verschiedene theoretische Lösungsansätze unter dem Dach einer einheitlichen Notation für den Anwender transparent anwenden lassen und in der Summe zu einer deutlichen Verbesserung der Software-Qualität in einem besonders komplexen Anwendungsgebiet beitragen können.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Die Entwicklung von Software hat einen schlechten Ruf. Viele Projekte scheitern, sprengen den geplanten Kostenrahmen oder führen zu Resultaten, die beim Kunden den Eindruck hinterlassen, er sei Teil des Reifeprozesses des Produktes. Komplexe Software, die ohne Fehler ausgeliefert wird, ist eher die Ausnahme als die Regel. Spektakuläre Software-Desaster wie die Zerstörung einer Ariane 5 Rakete aufgrund eines einfachen Konvertierungsfehlers oder die verspätete Einführung des deutschen Mautsystems finden in der breiten Öffentlichkeit große Aufmerksamkeit. Der wirtschaftliche Schaden und Imageverlust durch Rückrufaktionen im Automobilbereich ist enorm.

Die Grund für den unbefriedigenden Zustand der allgemeinen Software-Qualität ist einfach: Die Erstellung von Programmen ist ab einer gewissen Größe überaus komplex und für den menschlichen Verstand kaum zu beherrschen. Wirtschaftliche Zwänge begrenzen den Zeitraum, der für eine Fehlerbereinigung zur Verfügung steht.

Dennoch gibt es eine große Klasse von Anwendungen, bei denen Fehlerfreiheit gefordert werden muss, da ein Versagen der Software fatale Folgen haben kann: Eingebettete Echtzeitsysteme steuern komplexe Maschinen, Werkzeuge, Fahrzeuge und Instrumente. Im medizinischen Bereich, in der Luftfahrt oder in bestimmten Bereichen der Automobiltechnik kann fehlerhafte Software Menschenleben kosten. Im Bereich von Robotersteuerungen können Implementierungsfehler zumindest einen großen finanziellen Schaden verursachen. Dabei sind die Herausforderungen an die Software-Entwicklung in diesen Bereichen oft noch wesentlich größer als in einfachen Desktop-Anwendungen.

Nehmen wir als Beispiel die Robotik: Neuere Forschungsentwicklungen

auf diesem Gebiet nähern sich Produktionsgeschwindigkeiten von $10\,m/s$ in kleinen Arbeitsräumen und mit hochkomplexen Kinematiken an.[1] Unter diesen Randbedingungen werden auch modernste Werkstoffe bis an die Grenze ihrer Leistungsfähigkeit belastet. Ein Fehler in der Software führt dann fast zwangsläufig zur spektakulären Zerstörung der Roboterstruktur. Mindestens ebenso gefährlich wie falsche Berechnungen sind Ergebnisse, die zu spät kommen. Um einen Roboter bei einer Geschwindigkeit von $10\,m/s$ kontrollieren zu können, wird ein Regeltakt von mindestens $8\,kHz$ benötigt. Sollte der jeweils nächste Sollwert nicht rechtzeitig vorliegen, kann dieses im ungünstigen Fall ebenfalls zu einer Zerstörung der Struktur führen. Gefürchtet sind *Deadlocks* und Überschreitungen von *Deadlines*. Sie sind wesentlich schwieriger in den Griff zu bekommen, als algorithmische Korrektheitsanforderungen. Algorithmen lassen sich separat testen. Ob alle zeitlichen Anforderungen erfüllt werden, lässt sich erst überprüfen, wenn alle Komponenten des Gesamtsystems ausimplementiert und integriert sind. Dieses ist aus Sicht des Software-Engineerings der denkbar schlechteste Zeitpunkt, da sich mit jeder Entwurfsstufe der Aufwand eines Redesigns vervielfacht.

Die Entwicklung von Echtzeitsystemen ist also wesentlich kritischer als die Entwicklung von anderen Anwendungen, da Fehler lebensbedrohliche Konsequenzen haben können. Sie ist komplexer, weil neben der Korrektheit der Algorithmen auch der zeitliche Determinismus eine Schlüsselrolle spielt. Als sei das nicht schon Herausforderung genug, werden Fehler oft erst dann sichtbar, wenn ihre Beseitigung am aufwendigsten bzw. teuersten ist. Wie kann dennoch die Sicherheit von Echtzeitanwendungen mit vertretbarem Aufwand verbessert werden?

Diese Frage wird innerhalb dieser Arbeit behandelt. Natürlich ist es nicht möglich, an dieser Stelle ein „silver Bullet“ zu liefern, das alle Probleme mit einem Schlag löst. Ziel ist es, Techniken zu entwickeln, die beim Aufspüren bestimmter Fehlerklassen helfen. Die Grundidee ist, zu einem sehr frühen Zeitpunkt in der Entwicklung anzusetzen, um Fehler möglichst kostengünstig beseitigen zu können. Dafür wird in Kauf genommen, dass diese frühen Analysen nicht alle Fehlerquellen ausschalten können. Es geht darum, Fehler im Software-Design aufzuspüren. Dabei spielen zeitliche Anforderungen eine zentrale Rolle.

In diese Arbeit sind langjährige, praxisnahe Erfahrungen aus dem Sonderforschungsbereich 562 „Robotersysteme für Handhabung und Montage“

[1] Siehe SFB 562, http://www.tu-braunschweig.de/sfb562

eingeflossen. Dieser Sonderforschungsbereich ist seit Juli 2000 an der TU Braunschweig angesiedelt und wird durch den Sprecher Prof. Friedrich Wahl vertreten. Im SFB 562 werden Roboter mit einer besonderen Struktur - den parallelen Kinematiken - erforscht und in Hinblick auf Präzision und Geschwindigkeit optimiert. Steuerungen dieser Roboter weisen extrem hohe Echtzeitanforderungen auf. Diesen Herausforderungen mit Mitteln des Software-Engineerings zu begegnen, ist Aufgabe des Teilprojektes „Softwaretechnik und formale Analyse“ unter der Leitung von Prof. Ursula Goltz. Aus diesem Teilprojekt sind viele Ideen zu Ansätzen entsprungen, die hier verwirklicht wurden.

1.1 Ansätze

In dieser Arbeit werden verschiedene Ansätze verfolgt. Die Motivation für die verschiedenen Ansätze entspringt den unterschiedlichen Problemstellungen, die bei der Entwicklung von Echtzeitsystemen auftreten. Im Folgenden wird ein Überblick über die Ansätze und die Kontexte, in denen sie eingesetzt werden, gegeben.

Wichtiges Kriterium für die Umsetzung der Analyse ist eine größtmögliche Benutzerfreundlichkeit. Dazu gehört, dass proprietäre Notationen weitgehend vermieden werden sollten. Wenn für eine Analyse aufwendig eigene Modelle erstellt werden müssen, ist die Gefahr groß, dass bei nachträglichen Modifikationen Modell und Realisierung auseinander laufen. Je enger eine Analyse auf einer Notation aufsetzt, die für den Entwurf sowieso zur Modellierung benutzt wird, desto größer wird die Chance, dass ein solches Auseinanderdriften nicht auftritt. Nur dann bleiben Analyseergebnisse aussagekräftig und auf den Entwurf übertragbar.

1.1.1 UML-basierte Analyse

Es stellt sich die Frage, auf welcher Basis Modelle von Echtzeitsystemen analysiert werden sollen. In Bereich der Softwaretechnik hat sich die *Unified Modeling Language (UML)* [OMG01] als Standardsprache für eine objektorientierte Modellierung etabliert. Sie ist historisch aus dem Zusammenschluss verschiedener Ansätze gewachsen und das Ergebnis eines zähen Ringens um die „beste“ Modellierungssprache. Mit ihr lassen sich Sichten auf ein Software-System in Form von vordefinierten Diagrammen erstellen. Insbesondere bie-

tet sie zahlreiche Diagramme zur Modellierung der Software-Dynamik (Aktivitäts-, Sequenz-, Kollaborations- und Zustandsdiagramme).

Alternativ dazu gibt es Modellierungssprachen, die auf den Echtzeitbereich zugeschnitten sind. Ein wichtiger Vertreter dieser Sprachen ist eine Notation, die in der Methodik ROOM [SGW94a] verwendet wird. Diese Notation weist sehr interessante Ansätze auf. Allerdings zeichnete sich ab, dass sich ROOM gegenüber der UML nicht lange als eigenständige Notation behaupten würde.[2] Gegen spezialisierte Sprachen spricht der geringe Bekanntheitsgrad und das Argument der Zukunftssicherheit.

Ausschlaggebend für die Verwendung der UML in dieser Arbeit waren die positiven Folgen der Standardisierung: eine weitreichende Verbreitung unter den Anwendern und eine große Anzahl von verfügbaren Werkzeugen. Mittlerweile gibt es mehr als 100 Anwendungen, die versprechen, UML konform zu sein. Vorteilhaft ist weiterhin, dass die UML Mechanismen enthält, die eine nahezu unbeschränkte Erweiterung auf der Basis von Stereotypen ermöglichen. Diese Erweiterungen können Lücken in der Aussagekraft schließen, sollten aber vorsichtig angewendet werden, wenn man Wert auf die Vorteile einer standardisierten Beschreibung legt.

Die UML enthält eine sehr große Menge von Konstrukten, die in Modellen angewendet werden können. Allerdings ist nicht vorgesehen, dass in einem Software-Projekt alle Konstrukte und Diagramme eingesetzt werden. Gleiches gilt für die Verwendung als Eingangssprache für eine Analyse von Echtzeitmodellen. Daher gilt es, in einem ersten Schritt festzulegen, welche Fragestellungen für einen Anwendungsbereich interessant sind und mit welchen Modellen diese Fragestellungen beschrieben werden.

Dabei hilft, dass bestimmte Diagrammarten sich oft in bestimmten Entwicklungsschritten wiederfinden. Diese Koppelung ist zwar nur lose, da die UML ausdrücklich nicht den Anspruch erhebt, Methodiken zu definieren, sondern sich auf den Aspekt der Notation beschränkt. Allerdings gibt es bestimmte typische Zuordnungen, die sich immer wieder finden lassen.

Weiterhin wird der Schwerpunkt dieser Betrachtung auf den dynamischen Diagrammarten liegen, da sie am besten geeignet sind, die Komplexität von Echtzeitsystemen zu erfassen, die sich vor allem durch eine unüberschaubare Vielfalt zeitlicher Abläufe und Zusammenhänge auszeichnen.

[2] Das Unternehmen ObjecTime, das das zu ROOM gehörende Werkzeug vertreibt, wurde frühzeitig von Rational aufgekauft. Das Werkzeug wurde daraufhin an die UML angepasst.

In Software-Entwicklungsprozessen werden oft Anwendungsfalldiagramme zuerst eingesetzt, insbesondere dann, wenn die Systemfunktionalität und deren Grenzen noch definiert werden müssen. Beinhalten Anwendungsfälle Kommunikationsabläufe zwischen Systemkomponenten, können sie durch Sequenzdiagramme präzisiert werden: Einzelne Anwendungsfälle können durch eine Sammlung von Beispielabläufen beschrieben werden, die definieren, wie Softwarekomponenten kooperieren, d.h. Nachrichten austauschen, um eine bestimmte Funktionalität zu bewirken.

Der Übergang von einer Projektidee hin zu einem Anwendungsfalldiagramm oder von dort zu einem System von Sequenzdiagrammen entzieht sich weitgehend einer Konsistenzanalyse, da Anwendungsfalldiagramme sehr informell und ohne präzise Semantik sind. Aus Analysesicht interessant wird es erst mit der Einführung von Sequenzdiagrammen zur Systemspezifikation, da deren Semantik deutlich besser zu fassen ist, als die von Anwendungsfällen oder gar die einer Projektidee.

Setzt man die Verfeinerung des Entwurfs fort, ergibt sich dann typischerweise in einem objektorientierten Ansatz ein Bruch: Einerseits werden dynamische Kommunikationsabläufe durch eine Menge von Szenarien spezifiziert. Gemäß ihrer Verwendung zur Präzisierung von Anwendungsfällen legen sie fest, was ein System zu leisten hat. Wir interpretieren sie daher als Mittel zur *Anforderungsbeschreibung*. Ein objektorientiertes System wird allerdings Szenarien, die sich über mehrere Objekte erstrecken und damit einer *Interobjekt-Sichtweise* entsprechen, nicht direkt implementieren.

Andererseits wird das interne Verhalten von Softwarekomponenten oft durch Zustände und Zustandswechsel beschrieben. Damit wird eine *Intraobjekt-Sichtweise* eingenommen. Für reaktive Systeme ist dieses eine Beschreibungsform, die von der Struktur her schon nahe am endgültigen Programmcode angesiedelt ist. Ein Indiz dafür ist, dass es mittlerweile Werkzeuge gibt, welche diesen Schritt hin zur Implementierung automatisch vollziehen. Wir interpretieren daher eine zustandsbasierte Beschreibung als *abstraktes Implementierungsmodell*.

Diese zwei Sichtweisen werden in der UML durch Sequenzdiagramme und Zustandsdiagramme unterstützt. Prinzipiell gibt es zwei Möglichkeiten, wie diese komplementären Sichtweisen konsistent gehalten werden können. Einerseits kann eine Beschreibung durch Szenarien genutzt werden, um eine zustandsbasierte Beschreibung für Objekte zu generieren. Dieses setzt voraus, dass die Spezifikation durch Szenarien das gewünschte Verhalten *vollständig* beschreibt. In der Regel ist das nicht der Fall, sondern es handelt sich eher

um eine Sammlung von Beispielabläufen, da eine vollständige Verhaltensspezifikation mit Sequenzdiagrammen aufwendig ist.

Wenn Sequenzdiagramme nur beispielhafte Abläufe beschreiben, kann andererseits geprüft werden, ob diese Szenarien zu dem abstrakten Implementierungsmodell, bestehend aus Zustandsdiagrammen, *konsistent* sind. Ob ein Sequenzdiagramm konsistent mit einem System von Zustandsmaschinen ist, hängt davon ab, welchen Stellenwert es zugeschrieben bekommt. Hierfür werden in dieser Arbeit Interpretationsmöglichkeiten herausgearbeitet, welche die weitere Überprüfung beeinflussen. Insbesondere wird unterschieden, ob spezifiziertes Verhalten eintreten muss oder nur möglich ist.

Es wird hier von der zweiten Alternative ausgegangen. Es wird der Ansatz verfolgt, dass der kreative Prozess des Zusammenführens von Anforderungen, die sich aus einer Sammlung von Szenarien ergeben, und deren Realisierung innerhalb von Objekten dem Anwender überlassen bleibt. Allerdings soll ihm ein intuitives Werkzeug an die Hand gegeben werden, welches das Ergebnis dieser Bemühungen automatisiert auf Korrektheit überprüft. Ziel ist also, den Sprung von einer Interobjekt-Sicht hin zu einer Intraobjekt-Sicht werkzeugbasiert abzusichern. Da dies vor allem in Hinblick auf Echtzeitsysteme passiert, werden zeitliche Anforderungen dabei besonders berücksichtigt.

Welche Techniken sind aber geeignet, um dieses Ziel zu erreichen? In Fallstudien des SFB 562 hat sich Model-Checking bei der Verifikation von zeitkritischen Softwaresystemen als ein wertvolles Hilfsmittel erwiesen. Leider zeigte sich schnell, dass die formalen Beschreibungssprachen, die von diesen Werkzeugen benutzt werden, zur Modellierung großer Systeme unzureichend sind. Insbesondere das Fehlen hierarchischer Konstrukte und die nicht standardisierte Notation bereiten Probleme. Aus den oben genannten Gründen wäre es wünschenswert, dass ein Anwender keine zusätzlichen Modelle erstellen muss, sondern stattdessen die Verifikation auf standardisierten, praxisnahen Notationen aufsetzt. Daher wird in dieser Arbeit die Anbindung an eine formale Verifikationsmethode für den Benutzer unsichtbar automatisiert durchgeführt.

1.1.2 Anwendungsspezifische Analyse

Werden mit der oben genannten Methode Fallstudien in einem Anwendungsbereich bearbeitet, stellt sich schnell heraus, dass bestimmte Konstrukte immer wieder auftreten, aber von der UML nicht direkt unterstützt werden. Beispielsweise ist Message-Passing ein gängiges Kommunikationsmittel zwischen

Prozessen. Zur Vermeidung einer Prioritätenumkehr wird in Echtzeitsystemen oft Priority-Inheritance implementiert. Beides ist nicht Teil der UML, muss aber bei einer Analyse semantisch exakt behandelt werden.

Im ersten Ansatz kann man diese Konstrukte ebenfalls mit UML modellieren. Allerdings entstehen dabei überfrachtete Modelle, da ein Konstrukt des Anwendungsbereiches nun durch einen ganzen Modellteil repräsentiert wird. Außerdem wirkt sich dieser Umweg auch auf eine Verifikation aus, da benutzte Hilfszustände, Variablen und Uhren den Zustandsraum vergrößern. Eine direkte Modellierung in einer formalen Zielsprache könnte Optimierungspotential besser ausnutzen.

Selbst wenn ähnliche Konstrukte in der UML schon vorhanden sind, findet man selten eine genau identische Semantik, wie sie die Konstrukte im Anwendungsbereich haben. Da auch kleine semantische Unterschiede bei einer formalen Verifikation in der Regel zu falschen Ergebnissen führen, werden sich oft umständliche Konstruktionen nicht vermeiden lassen, wenn man diese Konstrukte auf UML-Ebene darstellt.

Ein besserer Weg ist, für anwendungsnahe Elemente Stereotypen einzuführen. Diese werden in der UML in Profilen gesammelt. Daher gilt es, zunächst den Anwendungsbereich auf zentrale Mechanismen hin zu analysieren, die bei einer Modellierung immer wieder benötigt werden. Ein UML-Profil stellt Zusammenhänge zwischen den neu eingeführten Modellelementen durch Klassendiagramme dar. Da diese Diagramme die Abhängigkeit in der Modellwelt beschreiben, werden sie als Domainmodelle bezeichnet. Die neuen Konstrukte werden durch Stereotypen in Modelle eingeführt. Ein Stereotyp ist eine Markierung für ein schon vorhandenes Modellelement, die es als etwas Besonderes mit modifizierter Semantik ausweist. Die Semantik von Stereotypen wird oft textuell beschrieben.

In dieser Arbeit sollen Echtzeitsteuerungen analysiert werden, die auf dem Echtzeitbetriebssystem QNX laufen. Da hierbei Konstrukte wie Threads, Prozesse und Message-Passing über Kanäle eine Rolle spielen, reicht der Sprachumfang der UML allein nicht aus. Es werden also in dieser Arbeit die wichtigsten Mechanismen in einem neuen Profil gesammelt, die für eine effiziente Modellierung vor dem Hintergrund der UML nötig sind.

In einem zweiten Schritt wird erläutert, wie sich derartige Modelle analysieren lassen. Da diese Analyse als Erweiterung und Spezialisierung der oben beschriebenen UML-basierten Analyse dynamischer Diagramme gedacht ist, wird insbesondere untersucht, ob sich ähnliche Verfahren wie bei der UML-basierten Analyse einsetzen lassen. Tatsächlich eignet sich auch hier Model-

Checking für die Durchführung der Analyse.

1.1.3 Schedulability-Analyse

Die oben beschriebene dynamische Analyse, sei es auf UML-Ebene oder anwendungsnah, ist in der Modellierung sehr flexibel. Es gibt allerdings zwei Fälle, in denen eine derartige Analyse nicht sinnvoll ist:

- Das System wird auf Hardware umgesetzt, die über genügend Reserven bezüglich der Rechenkapazität verfügt. Dann reicht eine pessimistische Abschätzung aus, welche die genauen dynamischen Abläufe außer Acht lässt, um Deadlines zu verifizieren.

- Es handelt sich um ein System, das aufgrund seiner Größe die Möglichkeiten einer aufwendigen Analyse aller dynamischen Abläufe sprengt. In diesem Fall kann eine Schedulability-Analyse das gewünschte Ergebnis liefern.

Für die Analyse der Schedulability gibt es in der Literatur eine Vielzahl von Verfahren. Dabei wird die Eigenschaft eines Systems untersucht, stets alle Deadlines einzuhalten. Allerdings wird in der Regel nur statisch analysiert. Es werden pessimistische Annahmen gemacht, damit ein positives Ergebnis auf jeden Fall Gültigkeit hat. Umgekehrt ist es möglich, dass ein Ablauf, der eine bestimmte Deadline verletzt, tatsächlich nie auftritt. Beispielsweise gehen viele Verfahren davon aus, dass es einen Zeitpunkt 0 gibt, an dem alle zyklisch arbeitenden Prozesse rechenwillig sind. In diesem Fall ergeben sich die schlechtesten Antwortzeiten. Wenn aber dies aufgrund von Phasenverschiebungen nie auftritt, handelt es sich um eine zu pessimistische Abschätzung.

Gerade wegen dieser Vereinfachung arbeiten solche Verfahren mit einem wesentlich geringerem Rechenaufwand. Mit Hilfe von Algorithmen aus der Literatur lassen sich aus statischen Eingaben mit geringem Rechenaufwand schnell gute Ergebnisse erzielen.

Es stellt sich auch hier die Frage, wie sich eine solche Analyse möglichst nahtlos in die UML integrieren lässt. Da der UML-Kern entsprechende Sprachkonstrukte nicht bereitstellt, muss wieder auf ein Profil zurückgegriffen werden. Das UML-Konsortium bietet ein standardisiertes Profil für Echtzeitsysteme an, das insbesondere auf die statische Analyse dieser Modelle zuge-

schnitten ist. Es bietet sich an, Begrifflichkeiten dieses Profils zu nutzen. Dieses Profil enthält eine umfangreiche Sammlung von Stereotypen, mit denen sich Schedulability-Aspekte modellieren lassen. Es enthält allerdings weder einen Hinweis darauf, wie diese Stereotypen in Modellen angewandt werden können, so dass diese analysierbar werden, noch wie eine solche Analyse in einem UML-Werkzeug umgesetzt werden kann.

Um das Profil sinnvoll einsetzen zu können, muss in einem ersten Schritt zunächst geklärt werden, welches konkrete Verfahren zur Analyse der Schedulability genutzt werden soll. Ein solches Verfahren enthält einerseits einen Algorithmus, der das Analyseergebnis ermittelt und andererseits einen Text, der den Kontext beschreibt, in dem dieses Verfahren eingesetzt werden kann. Ausgehend von dieser Beschreibung ist zu klären, welche Entitäten in einem analysierbaren Modell vorkommen müssen und wie diese Entitäten zu verknüpfen sind.

Im UML-Modell werden diese Entitäten durch Stereotypen repräsentiert. Hier ist zu definieren, welche UML-Elemente dafür in Frage kommen. Dafür sind Fragen zu beantworten wie „Kann ein Trigger in UML durch ein Ereignis oder durch eine Transition dargestellt werden?“. Im vorgegebenen Profil werden die Anwendungsmöglichkeiten von Stereotypen zwar schon eingeschränkt, allerdings ist eine weitere Begrenzung im Kontext eines konkreten Analyseverfahrens nötig.

Letztlich bleibt zu definieren, wie aus einem stereotypisierten Modell die Parameter für eine Analyse extrahiert werden können. Dabei muss vor allem geklärt werden, wie die Verknüpfungen zwischen Entitäten aus dem Modell rekonstruiert werden können. In dieser Arbeit werden diese Schritte exemplarisch für einen weit verbreiteten Analysealgorithmus untersucht.

In dieser Arbeit werden also drei Ansätze verfolgt, um die Entwicklung von Echtzeitsystemen zu unterstützen. Gemeinsam haben sie die benutzerfreundliche Integration von formalen Methoden in die UML-Modellwelt. Wie aber grenzen sich die Ansätze von einander ab, und warum reicht nicht ein Ansatz aus? Jeder von ihnen bezieht sich auf eine bestimmte Abstraktionsebene und ist dazu geeignet, unterschiedliche Fehlerklassen aufzuspüren. In Abbildung 1.1 werden die Verfahren Entwicklungsstufen im V-Modell zugeordnet.

Der erste hier beschriebene Ansatz zielt darauf ab, die Korrektheit eines Entwurfs des internen Verhaltens von Objekten anhand von Beispielszenarien zu überprüfen. Die Abstraktionsebene ist relativ hoch und daher wird auf die

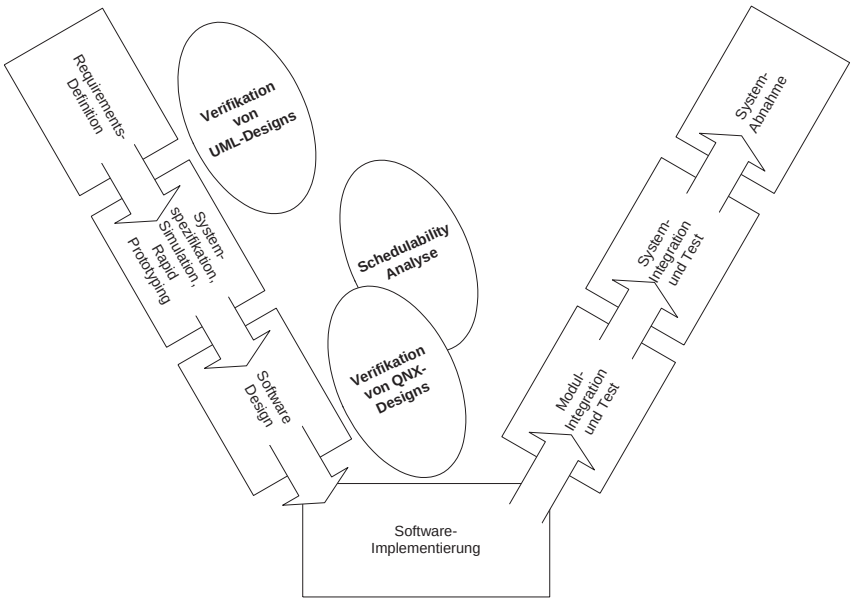


Abbildung 1.1: Ansätze für eine Werkzeugunterstützung

Modellierung von implementierungsspezifischen Details weitgehend verzichtet. Im V-Modell würde sich dieser Ansatz am Übergang von Requirements-Definition und Systemspezifikation wiederfinden. Allerdings werden schon in dieser Entwicklungsstufe Möglichkeiten zur Modellierung von zeitlichen Aspekten benötigt, da für Echtzeitsysteme die Berücksichtigung von Zeit schon in frühen Phasen erforderlich ist. Die Semantik der Modellkonstrukte orientiert sich am UML-Standard. Grenzen sind diesem Ansatz gesetzt, wenn eine implementierungsnähere Sicht eingenommen werden soll.

Der zweite Ansatz berücksichtigt implementierungsspezifische Konstrukte, die in einem eigenen Profil zusammengefasst werden. Deren Semantik ist durch den Anwendungsbereich vorgegeben. Wir orientieren uns in dieser Arbeit an Konstrukten, die dem Echtzeitbetriebssystem QNX entlehnt sind. Analysiert werden das Zusammenspiel von Kommunikationsmechanismen, Scheduling und Thread-Verhalten. Die Analyseergebnisse sind in diesem Fall spezifisch an einen Anwendungsbereich gebunden, was eine vertiefte Analyse erlaubt, aber der Übertragbarkeit von Ergebnissen Grenzen setzt. Im V-Modell wäre dieser Ansatz beim Übergang vom Software-Design zur Implementierung zu finden.

Der dritte Ansatz nutzt die Vorteile einer statischen Analyse. In diesem Fall wird auf ein standardisiertes Profil für den Echtzeitbereich aufgesetzt. Die Analyseergebnisse sind generell „gröber“ als bei den bisher besprochenen Verfahren, da einzelne dynamische Abläufe nicht berücksichtigt werden. Oft werden in diesem Fall pessimistische Abschätzungen vorgenommen. Der Vorteil dieses Ansatzes ist sein sparsamer Umgang mit Rechnerressourcen, so dass auch die Analyse von sehr großen Modellen keine Probleme bereiten sollte. Der Abstraktionsgrad liegt zwischen den beiden oberen Verfahren, da zwar schon für Echtzeitsysteme typische Strukturen wie zum Beispiel Prioritäten direkt unterstützt werden; daher ist dieses Verfahren spezieller als der allgemeine UML-Ansatz. Allerdings werden noch keine Eigenschaften eines speziellen Echtzeitbetriebssystems berücksichtigt, so wie es beim zweiten Ansatz der Fall ist.

1.2 Literatur

Im Bereich des Model-Checkings gibt es eine Fülle von Ansätzen, die sich mit der Verifikation sicherheitskritischer Systeme beschäftigen. Gerade in jüngster Zeit wurde dabei eine gewisse Reife erreicht, die auch in populärwissenschaftlichen Zeitschriften Aufmerksamkeit erregt. So beschreibt [Klo05] die Anwendung von formalen Techniken bei der Entwicklung von Intel Prozessoren und der Überprüfung von Gerätetreibern.

In Deutschland gibt es mit dem vom Bundesministerium für Bildung und Forschung mit 3,5 Millionen Euro geförderten Projekt Verisoft[3], das seit dem 1. Juli 2003 läuft und auf 8 Jahre angelegt ist, und dem Sonderforschungsbereich/Transregio 14 AVACS[4] gleich zwei große Forschungsprojekte in diesem Bereich. Diese Projekte beschränken sich allerdings nicht auf Model-Checking, sondern untersuchen allgemein die Möglichkeiten formaler Techniken, wie auch Theorembeweisern und der abstrakten Interpretation von Programmcode. Diese Beispiele belegen, dass formale Techniken mittlerweile praxisnah eingesetzt werden können.

Verifikation dynamischer UML-Modelle. Grundlage der Verifikation sind Zeitautomaten, deren Theorie in [AD94, AD96] beschrieben wird. Die

[3]Projektseite http://www.verisoft.de/
[4]Projektseite http://www.avacs.org/

theoretischen Erkenntnisse führen in [LPY97] und [Yov97] zur Entwicklung der Werkzeuge UPPAAL und Kronos.

In der Literatur findet sich mit [KMR02] ein Ansatz, welcher der Analyse von dynamischen Modellen dieser Arbeit am meisten ähnelt. Dort wird der Vorschlag zur Abbildung von Sequenzdiagrammen in Zeitautomaten aus [FHD+99] übernommen und mit einer eigenen Konstruktion zur Abbildung von Zustandsdiagrammen versehen. Da [FHD+99] nur für eine synchrone Kommunikation auf Ebene von Zeitautomaten gedacht ist, bleibt unklar, wie sich dieser Ansatz mit einer asynchronen Kommunikation in UML verbinden lässt, die vor allem auf Kommunikation über Ereignisse setzt. Daher wird hier [FHD+99] in Abschnitt 3.3 für den asynchronen Fall erweitert. Anders als in der vorliegenden Arbeit werden in Hugo/RT [KMR02] für alle Nachrichten maximale Laufzeiten angenommen und eine Verifikation auf die korrekte Abfolge von Nachrichten beschränkt. Sender und Empfänger werden bei der Verifikation nicht mit einbezogen.

In [DMY02] werden UML-Zustandsdiagramme in Zeitautomaten umgewandelt. Allerdings wird dort der Schwerpunkt auf die Einführung hierarchischer Konstrukte gelegt. Kommuniziert wird wie in Zeitautomaten über synchrone Aktionen. Ereignisse, die zwischen Objekten ausgetauscht werden, werden nicht berücksichtigt.

Ansonsten gibt es verschiedene Ansätze zur Analyse von Zustandsdiagrammen, die auf der Statemate Semantik [HN96] basieren, wie z.B. [MLS97]. Diese Semantik entstammt nicht dem objektorientierten Kontext, sondern der strukturellen Analyse und Ergebnisse lassen sich nicht direkt auf die UML übertragen.

Verifikation von anwendungsspezifischen Modellen. Für diesen Teil der Arbeit ist einerseits eine Modellierung in UML und andererseits eine sehr nahe Ausrichtung an dem speziellen Anwendungskontext des Echtzeitbetriebssystems QNX kennzeichnend. In diesem Bereich sind uns keine unmittelbar verwandten Arbeiten bekannt. In [AFM+03, FMPY03] werden Echtzeitsysteme modelliert und deren dynamisches Verhalten analysiert. Bezüglich der angewendeten Verifikationstechnik basiert auch dieser Ansatz auf Model-Checking von Zeitautomaten. Das zugehörige Werkzeug benutzt allerdings eine proprietäre graphische Notation.

Schedulability-Analyse auf Basis des standardisierten UML-Profils. Bezüglich einer Schedulability-Analyse gibt es eine umfangreiche Liste von Forschungsansätzen. Beispielhaft seien [LL73, SRS94, LRD99] genannt. Einen Überblick über gebräuchliche Algorithmen geben [BW01, Kop97].

Um die Analyse der Schedulability anwenderfreundlich zu gestalten, existieren Werkzeuge, welche die Eingabe der benötigten Informationen mit graphischen Modellen verbinden [AFM+02, BP99]. Entsprechende Werkzeuge unterstützen proprietäre graphische Modellierungssprachen, die sich nicht mit UML-Modellen verbinden lassen.

Da die UML ohne Erweiterungen zu ausdrucksschwach ist, um eine Analyse der Schedulability zu erlauben, ist ein Profil notwendig, das benötigte Konstrukte definiert. Die Standardisierungskommission OMG[5] hat mit [Obj05] die erforderlichen Erweiterungen für den Echtzeitbereich festgelegt. Dessen erste offizielle Version wurde im Jahre 2001 herausgegeben.

1.3 Aufbau dieser Arbeit

Im Kapitel 2 wird zunächst der Frage nachgegangen, wie sich Echtzeitaspekte modellieren lassen. Mit der UML lassen sich gleiche Sachverhalte auf verschiedenste Weise ausdrücken. Allerdings muss für eine automatisierte Analyse präzise beschrieben werden, wie analysierbare Modelle aufgebaut sein müssen. Dazu werden in Abschnitt 2.1 und 2.2 Modellierungssprachen auf Basis von Zustands- und Sequenzdiagrammen entwickelt.

Sollen anwendungsspezifische Konstrukte untersucht werden, müssen spezialisierte Stereotypen einführt werden. Diese werden in einem UML-Profil zusammengestellt. Dieses Vorgehen wird im zweiten Teil des Kapitels im Abschnitt 2.3 beispielhaft für den Anwendungsbereich von QNX-Anwendungen durchgeführt. Dort wird die dynamische Modellierung durch eine Sicht auf die Systemarchitektur ergänzt.

Die Modelle, die im zweiten Kapitel eingeführt werden, haben den Anspruch, automatisiert analysierbar zu sein. Entsprechende Verfahren werden in Kapitel 3 vorgestellt. Zunächst werden in 3.1 Zeitautomaten eingeführt, auf denen die Analyse aufgebaut wird. Danach wird jeweils die Transformation von Zustandsdiagrammen (Abschnitt 3.2), Sequenzdiagrammen (Abschnitt 3.3) und anwendungsspezifischen Modellen (Abschnitt 3.4) in diesen Formalismus im Detail beschrieben. Die Reihenfolge der Behandlung von Sequenz-

[5] Object Management Group

und Zustandsdiagrammen ist in diesem Kapitel vertauscht, weil die erste Transformation auf der zweiten aufsetzt.

Kapitel 4 widmet sich der Schedulability-Analyse. Da hier eine grundlegend andere Technik zum Tragen kommt, wird dieser Ansatz getrennt von den vorherigen Verfahren behandelt.

Diese Arbeit endet mit zusammenfassenden Bemerkungen und einem Ausblick in Kapitel 5.

Kapitel 2

Modellierung von Echtzeitsystemen

In diesem Kapitel wird beschrieben, wie sich Echtzeitsysteme mit Hilfe der *Unified Modeling Language* (UML) spezifizieren lassen. Dabei wird der Schwerpunkt auf eine dynamische Modellierung gelegt, da gerade hier Werkzeuge, die eine automatische Analyse ermöglichen, einen besonderen Nutzen haben. UML ist eine sehr mächtige Sprache. Wenn man zusätzlich noch die Möglichkeiten in Betracht zieht, die sich aus ihren Erweiterungsmechanismen ergeben, lassen sich Sachverhalte auf sehr unterschiedliche Weise beschreiben. In diesem Kapitel geht es vor allem darum, diese Möglichkeiten so weit einzugrenzen, dass eine automatische Analyse realisierbar wird. Richtschnur für die Sprachdefinitionen sind vor allem Erfahrungen, die innerhalb eines praktisch ausgerichteten Projektes[1] aus dem Bereich der Robotik gewonnen wurden.

Ein zu entwickelndes Softwaresystem wird in dieser Arbeit aus drei Sichten betrachtet. Es werden Sprachen zur Modellierung einer *Anforderungssicht,* einer *Implementierungssicht* und einer *Sicht auf das Scheduling* zur Verfügung gestellt. Die ersten beiden Sichten beschreiben Interobjektverhalten (Wie kollaborieren Objekte durch den Austausch von Nachrichten?) und Intraobjektverhalten (Welches Verhalten weisen Objekte auf?). Sie müssen zueinander konsistent sein. Die dritte Sicht wird isoliert betrachtet. Hier wird eine Sicht noch näher an der Implementierung eingenommen, um einen möglichst intuitiven Zugang zu der komplexen Welt der Schedulability-Analyse

[1] Sonderforschungsbereich 562 „Robotersysteme für Handhabung und Montage“

zu schaffen. Dieser Bereich wurde bisher vor allem durch nicht standardisierte Modellierungssprachen beherrscht.

2.1 Anforderungssicht

In diesem Abschnitt wird die Anforderungssicht auf Software behandelt. Die UML stellt verschiedene Diagrammtypen zur Verfügung, die Software aus dieser Sicht beschreiben. Beispielsweise dienen Anwendungsfalldiagramme dazu, eine grobe Übersicht über Funktionalitäten und deren Zusammenhänge sowie ihre Verbindungen zur Umgebung auf einfache Weise zu visualisieren. Allerdings eignen sich Anwendungsfalldiagramme nicht für eine weitergehende Analyse auf formaler Basis, da eine semantische Formalisierung fehlt und auch nicht sinnvoll erscheint.

Aktivitätsdiagramme hingegen eignen sich für eine Formalisierung und es lassen sich in der Literatur entsprechende Ansätze finden [EW01]. Sie werden oft in Entwicklungsprozessen eingesetzt, um Anwendungsfälle in Einzelaktionen zu unterteilen. So verwendet, fehlt ihnen ein Bezug zu einem objektorientierten Entwurf und sie sind daher ungeeignet, um eine objektbasierte Verhaltensbeschreibung zu verifizieren.[2]

In der UML erweisen sich Sequenzdiagramme als besonders nützlich, um Verhaltensanforderungen bezüglich eines objektorientierten Entwurfs zu beschreiben. Sie enthalten durch die Darstellung von Instanzen einen direkten Bezug zu einer objektorientierten Betrachtung von Software, so dass sich ein guter Ausgangspunkt für weitere Analysen ergibt. Weiterhin eignen sie sich sehr gut für eine Präzisierung von Anwendungsfällen, d.h. man nimmt eine Sichtweise ein, die sich an den Anforderungen eines zu erstellenden Systems orientiert. Im Folgenden werden Sequenzdiagramme, wie sie in dieser Arbeit verwendet werden, im Detail vorgestellt.

2.1.1 Sequenzdiagramme

Mit Sequenzdiagrammen lassen sich vor allem Abfolgen von Nachrichten und Ereignissen, und damit die Interaktion zwischen Objekten in anschaulicher

[2] In späteren Entwurfsphasen können mit Hilfe von *Swimlanes* Hinweise auf eine Zuordnung von Funktionalitäten zu Objekten integriert werden. Dieser Ansatz wurde hier nicht verfolgt.

Weise darstellen. Um ein brauchbares Hilfsmittel für eine Beschreibung zu erhalten, die ausreichend ausdrucksstark ist, sind zwei Aspekte entscheidend.

- Für Echtzeitsysteme sind zeitliche Anforderungen essentiell. Um maximale Ausführungszeiten von Kommunikationsfolgen zu modellieren, sind in der UML schon ausreichende syntaktische Elemente vorhanden, auf die zurückgegriffen wird.

- Notwendigerweise muss die Interpretation eines Sequenzdiagramms festgelegt werden. Wie bei Harel und Damm in [DH99] zur Motivation von *Life Sequence Charts (LSC)* ausgeführt wird, fehlt dieses Konzept in der UML bisher völlig. Es kann nicht modelliert werden, ob ein Sequenzdiagramm obligatorisch ist oder nur erwünschtes Verhalten darstellt. Sehr hilfreich ist die Modellierung von Vorbedingungen (z. B. eine Ausnahmesituation) und darauf folgende Reaktionen.

Im Folgenden wird beschrieben, welche Konstrukte innerhalb von Sequenzdiagrammen für diesen Ansatz benötigt werden. Dabei wurde die Auswahl einerseits davon bestimmt, dass die unterstützten Modellelemente ausdrucksstark genug sein müssen, um praxisnahe Probleme mit minimalem Aufwand zu modellieren. Sequenzdiagramme zur Modellierung von Echtzeitsystemen sollten sich mit gängigen Modellierungswerkzeugen erstellen lassen und einem UML-Kundigen eine intuitive Benutzung erlauben. Dort, wo eine Erweiterung des UML-Kerns zwingend notwendig ist (siehe oben), sollte sie sich mit Hilfe der Erweiterungsmechanismen des Standards realisieren lassen. Ansonsten würden sich die Vorteile der UML - große Benutzerakzeptanz und Werkzeugunterstützung - schnell relativieren.

Andererseits soll die eigentliche Verifikation ohne weitere Benutzerinteraktion ablaufen. Daher dürfen die als Eingabesprache benutzten Sequenzdiagramme nur Konstrukte erlauben, die sich im Verifikationswerkzeug nachbilden und den zu erwartenden Zustandsraum bei der Verifikation nicht übermäßig groß werden lassen. Das führt zu Kompromissen bei der Sprachdefinition, die erst im folgenden Kapitel, das sich mit der Analyse von Modellen beschäftigt, verständlich werden.

Hinweis: Die Entwicklung der UML ist längst noch nicht abgeschlossen. Zum Zeitpunkt der Erstellung dieses Dokuments war die Version 1.5 aktuell.[3]

[3] http://www.omg.org/technology/documents/formal/uml.htm

Soweit keine andere Version erwähnt wird, beziehen sich alle Angaben auf diese Version. Die Nachfolgeversion UML 2.0 konnte in dieser Arbeit nicht berücksichtigt werden. Soweit es zum jetzigen Zeitpunkt erkennbar ist, ändern sich die hier relevanten Konstrukte jedoch kaum oder gar nicht. Die massiven Änderungen bezüglich der dynamischen Modellierung betreffen vor allem die diagrammübergreifende Verknüpfung von Modellen. Die Entwicklung einer umfassenden Semantik ist aktueller Forschungsgegenstand.

2.1.1.1 Syntax

Sequenzdiagramme visualisieren die Kommunikation zwischen Instanzen oder Objekten eines Softwaresystems. Dabei wird die zeitliche Abfolge besonders hervorgehoben. Die beteiligten Instanzen werden horizontal angeordnet als Rechtecke dargestellt. Wahlweise kann ein Instanzname oder ein Typ durch einen Doppelpunkt abgetrennt angegeben werden. Entsprechende Bezeichnungen werden unterstrichen, um den Instanzcharakter hervorzuheben. Anders als in einem Objektdiagramm finden Beziehungen zwischen Objekten keine Berücksichtigung.[4]

Der zeitliche Verlauf wird von oben nach unten verstreichend angenommen, d.h. eine weiter unten im Diagramm dargestellte Nachricht folgt auf eine Nachricht, die sich weiter oben befindet. In diesem Verlauf werden Objekte durch *Lifelines* repräsentiert, von denen zugehörige Nachrichten ausgehen. Nachrichten werden durch Pfeile symbolisiert, die mit einem Nachrichtennamen beschriftet sein können.

Dieser Grundaufbau entspricht dem von verwandten Diagrammarten, wie beispielsweise den (älteren) *Message Sequence Charts* (MSC). In Abbildung 2.1 werden ein UML-Sequenzdiagramm (links) und ein MSC (rechts) gegenübergestellt. MSC haben aufgrund ihrer Formalisierung und der verbreiteten Anwendung innerhalb der Telekommunikation eine besondere Bedeutung. Allerdings erlauben sie nur eine nicht blockierende, asynchrone Kommunikation.

Die UML erhält viele Konstrukte, mit denen dieses Grundmodell verfeinert werden kann. In UML-Notation werden *synchrone* und *asynchrone Nachrichten* anhand ihrer Pfeilspitzen unterschieden: Synchrone Nachrichten weisen eine volle Pfeilspitze an einem horizontalen Pfeil auf, wohingegen asynchrone Nachrichten durch eine halbe Spitze gekennzeichnet sind. In diesem Fall kann der Nachrichtenpfeil nach unten geneigt sein.

[4] Ist dieser Aspekt wichtig, stellt ein Kollaborationsdiagramm eine gute Alternative da.

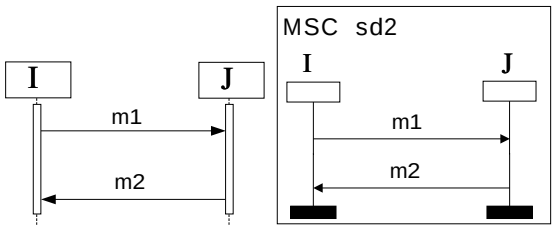


Abbildung 2.1: Sequenzdiagramm in der UML und als MSC

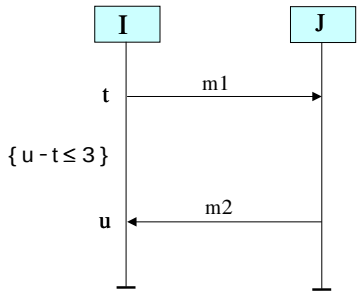


Abbildung 2.2: Sequenzdiagramm mit Zeitbedingung

Das Sende- und Empfangsereignis einer Nachricht sind implizit mit einem Zeitstempel versehen. Die Methoden *m.receiveTime* bzw. *m.sendTime* erlauben den Zugriff auf diese Zeiten (m ist ein Nachrichtenname). Alternativ dazu können bestimmte Ereignisse mit Marken versehen werden, die sich auf den Auftrittszeitpunkt beziehen und in Ausdrücken verwendet werden können. Diese Schreibweise ist ebenfalls UML-konform und wird hier bevorzugt angewendet, da sie eine kompaktere Schreibweise ermöglicht.

In der UML können *Constraints* in geschweiften Klammern an beliebiger Stelle in ein Diagramm platziert werden. Diese Schreibweise kann genutzt werden, um *zeitliche Anforderungen* zu formulieren, indem Zeitmarken in Beziehung zueinander gesetzt werden. So spezifiziert etwa die Bedingung $\{|b - a| \leq 10\}$, dass die zeitliche Differenz der durch a und b markierten Ereignisse maximal 10 Zeiteinheiten beträgt. In Abbildung 2.2 wird ein Beispiel für ein Sequenzdiagramm mit Echtzeitannotationen gegeben.

Sequenzdiagramme können entweder einen einzelnen Durchgang beschreiben oder mehrere erlaubte Abfolgen, indem Schleifen und Verzweigungen benutzt werden. In der UML wird in diesem Zusammenhang vom Gebrauch als *Instance* bzw. *Generic Form* gesprochen.

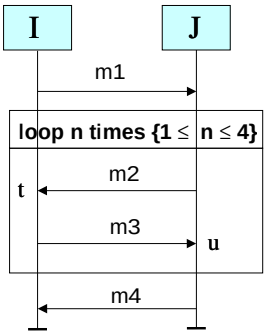


Abbildung 2.3: Sequenzdiagramm mit Schleife

Eine Schleife wird in einem Sequenzdiagramm durch ein Rechteck gekennzeichnet, das betroffene Nachrichten und Instanzen umschließt (Abb. 2.3). Die möglichen Schleifeniterationen lassen sich durch die Verwendung eines UML-Constraints einschränken. Um die Schleifenbedingung prägnant darzustellen, wird sie in dieser Arbeit in das Schleifenrechteck eingetragen. Diese Darstellung wurde von [SvG98] übernommen. Die Syntax der Schleifenbedingung beruht auf [FHD+99] und hat die Form $n\,times\ \{C_L\,(n)\}$. Ist $C_L\,(n)$ erfüllt, ist n als Iterationszahl erlaubt.

Werden für Nachrichten in Schleifen Zeitmarken vergeben, ist nicht definiert, auf welche Iteration sich eine Marke bezieht. Daher wird für eine Marke a ein a_{first} eingeführt, das sich auf das erste Auftreten von a innerhalb der Schleife bezieht. Dementsprechend bezieht sich a_{last} auf das letzte Auftreten von a in der Schleife und a_{next} auf die folgende Iteration. Sind a_{first}, a_{last} und a_{next} nicht definiert, weil beispielsweise eine Schleife nicht durchlaufen wird, wird ein Ausdruck, der diese Marken enthält, nicht berücksichtigt.

In der UML finden sich noch weitere Konstrukte, die hier nicht beschrieben werden, da sie für diese Arbeit nicht relevant sind. Nicht behandelt werden beispielsweise Aktivitätsperioden, Erzeugung/Zerstörung von Instanzen, etc. Für eine vollständige Sprachbeschreibung wird auf den Standardtext verwiesen.

Erweiterte Sequenzdiagramme. In der UML ist für Sequenzdiagramme nicht festgelegt, wie ein Diagramm zu interpretieren ist. In den ersten Iterationen eines Entwicklungsprozesses ist es sinnvoll, mit ihnen Beispielabläufe zu modellieren. Diese Beispielabläufe sind typischerweise mit Anwendungsfällen verknüpft. Ohne dass schon festgelegt ist, wie sich ein bestimmtes Sze-

nario in den Kontext anderer Anwendungsfälle einfügt, wird modelliert, dass ein Softwaresystem dieses Verhalten realisieren muss. Das heißt, wenigstens ein denkbarer Ausführungspfad wird das repräsentierende Sequenzdiagramm nachvollziehen.

In späteren Phasen wird diese Darstellung weiter verfeinert. Man wird spezifizieren, dass ein bestimmtes Verhalten als Reaktion auf vorhergehende Kommunikation auftreten *muss*. Sowohl die Vorbedingung, als auch die Reaktion kann als Sequenzdiagramm beschrieben werden. Es gibt viele Beispiele, bei denen eine solche Art der Modellierung sinnvoll ist. Beispielsweise kann als Vorbedingung ein Szenarium festgelegt werden, das durch die Betätigung einer Taste oder die Identifizierung einer Notfallsituation ausgelöst wird. Die Behandlung dieser auslösenden Abläufe wird dann im abhängigen Diagramm spezifiziert.

Letztlich gibt es Systeme, bei denen eine vollständige Beschreibung des Verhaltens erfolgen kann. Dieses ist insbesondere bei zyklisch ablaufenden Programmen der Fall. Die oben eingeführten Schleifenkonstrukte ermöglichen eine Modellierung von nicht terminierendem Verhalten. Diese Beschreibungsart scheint auf den ersten Blick relativ unflexibel, wenn man annimmt, dass ein System exakt nur die eine spezifizierte Nachrichtenabfolge ausführen kann. Bisher wurde allerdings noch nicht definiert, wie Nachrichten, die nicht im Sequenzdiagramm vorhanden sind, behandelt werden. Treten im System Nachrichten auf, die nicht in einem Sequenzdiagramm erscheinen, können sie entweder als Verletzung dieser Spezifikation gewertet oder ignoriert werden. Im Folgenden können beide Alternativen zur Modellierung genutzt werden. Dazu wird davon ausgegangen, dass jedes Diagramm mit einer Liste von *relevanten* Nachrichten (bezogen auf deren Namen) versehen ist. Nachrichten, die nicht in dieser Liste enthalten sind, können beliebig auftreten, ohne dass dies einen Einfluss auf die Bewertung des dazugehörigen Diagramms hat. Nachrichten, die in einem Diagramm verwendet werden, gelten implizit als relevant.

Um die Gebrauchsweise von Sequenzdiagrammen festlegen zu können, wird ein *Spezifikationsstatus* eingeführt. Diese Erweiterung des UML-Standards lässt sich problemlos mit den UML-eigenen Erweiterungsmechanismen realisieren. Die Semantik orientiert sich an der Kennzeichnung von Charts als *hot* oder *cold* aus [DH99]. *Hot* bedeutet dort, dass alle Durchläufe des Systems das Chart erfüllen müssen. *Cold* fordert, dass wenigstens ein Durchlauf des Systems existiert, der das Chart erfüllt. Nachrichten mit dem Status *cold* sind erlaubt, müssen aber nicht auftreten. Das *if ... then* Konstrukt

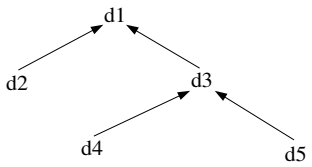


Abbildung 2.4: Bedingte Sequenzdiagramme

dieser Arbeit ist vergleichbar mit einer Bedingung aus [DH99], die als *cold* gekennzeichnet ist. Wenn eine solche Bedingung nicht erfüllt ist, wird das aktuelle Chart verlassen und die Überprüfung von nachfolgenden Charts abgebrochen.

- Der Status *Optional* für ein Sequenzdiagramm bedeutet, dass ein System die Anforderung erfüllt, wenn ein spezifizierter Ablauf möglich ist. Es ist aber nicht notwendig, dass er in jedem Durchlauf auftritt. Es wird also verlangt, dass ein System einen Ablauf zulässt, der konform zum Sequenzdiagramm ist (unter Berücksichtigung von Zeit- und Schleifenbedingungen) und vollständig bis zum Ende durchlaufen wird.

- Der Status *Mandatory* kennzeichnet Verhalten, das auftreten *muss*. Jede abweichende Kommunikation wird als Verletzung der Spezifikation gewertet. Dieses trifft allerdings nur auf Nachrichten zu, die im Diagramm erscheinen oder explizit als relevant markiert werden (siehe oben).

- Zwei Diagramme können durch ein *if ... then* Konstrukt verbunden werden. In diesem Fall wird ein Diagramm mit *Prechart* markiert und als Vorbedingung interpretiert. Der Spezifikationsstatus enthält dann Verweise auf weitere Diagramme, die als *Mandatory* gekennzeichnet sind oder weitere Verzweigungen beinhalten. Es wird gefordert, dass für alle Pfade im spezifizierten System ein bestimmtes Verhalten zwingend auftreten muss, wenn die Vorbedingung erfüllt ist.

Mit Hilfe des Status *Prechart* lassen sich Graphen von abhängigen Sequenzdiagrammen erzeugen (Abb. 2.4). Für diese Graphen wird gefordert, dass sie aus disjunkten Untergraphen mit einer Baumstruktur bestehen.

Zum Teil ergeben sich weitere Einschränkungen bezüglich der Modelle im Hinblick auf die Analysierbarkeit. Beispielsweise sind Zeitbedingungen, die

sich auf unterschiedliche Schleifendurchgänge beziehen, problematisch, wenn sich diese Schleife am Ende des Diagramms befindet. Es ist dann nicht zu entscheiden, ob ein System beim Ausbleiben weiterer Nachrichten schon terminiert hat oder die Wartezeit bis zur nächsten Nachricht nur ungewöhnlich lang ist. Diese Einschränkungen werden an entsprechender Stelle im Zusammenhang mit der Beschreibung der Analyseverfahren begründet.

Die Syntaxbeschreibung der hier verwendeten Sequenzdiagramme wird in den folgenden Definitionen zusammengefasst.

Definition. Eine *Anforderungsspezifikation* ist ein Tupel $\langle I, M, SD \rangle$ mit einer Menge I von *Instanzen*, einer Menge M aller *Nachrichten* eines Systems und einer Menge von *Sequenzdiagrammen* SD. Ein Sequenzdiagramm $sd \in SD$ ist ein Tupel $\langle M_{sd}, b_{sd}, C_{sd}, st \rangle$, mit einer Menge *relevanter Nachrichten* $M_{sd} \subseteq M$ und einem *Rumpfes* b_{sd}. Es enthält die zeitlichen Anforderungen C_{sd} und einen Status $st_{sd} \in \{Optional, Mandory, Prechart\}$.

Ein Rumpf $b \in B_{sd} = \{(m|l)^* \mid m \in M_{sd}, l \in L_{sd}\}$ eines Sequenzdiagramm besteht aus einer Abfolge von Nachrichten und Schleifen. Eine Schleife aus $L_{sd} = \langle b_{sd}, c_L \rangle$ enthält wiederum einen Rumpf aus B_{sd} und eine Schleifenbedingung c_L. Eine Schleifenbedingung c_L hat die Form $c_1 \triangleleft n \triangleleft c_2$ oder $n = c_1 \mid \ldots \mid c_n$, wobei die Iterationszahlen $c_1, \ldots, c_n$ nicht negative ganze Zahlen oder ∞ sind und $\triangleleft$ einer der Vergleichsoperatoren $<$ oder $\leq$ ist. Durch Einsetzen von 0 bzw. ∞ lassen sich damit auch Bedingungen wie $n < c_1$ oder $n \geq c_2$ angeben.

In der obigen Definition wird von einer totalen Ordnung auf Nachrichten innerhalb eines Rumpfes ausgegangen. Damit wird *nicht* vorweggenommen, dass auch Sende- und Empfangsereignisse total geordnet sind. Wird beispielsweise die visuelle Ordnung (vgl. 2.1.1.2) zugrunde gelegt und teilen sich zwei Nachrichten weder Sende- noch Empfangsinstanzen, wird durch ihre Anordnung im Rumpf *nicht* ihre zeitliche Ausführungsreihenfolge festgelegt. Eine partielle Ordnung auf Ereignissen ergibt sich in diesem Fall durch eine instanzweise Ordnung der Ereignisse und durch Beachtung der Abhängigkeit zwischen Sende- und Empfangsereignissen.

Mit der folgenden Definition wird die Verknüpfung einer Anforderungsspezifikation mit einem System vorbereitet.

Definition. Eine *systembezogene Anforderungsspezifikation* ist eine Anforderungsspezifikationen, die durch die Funktionen $Part : I \mapsto O$ und Com :

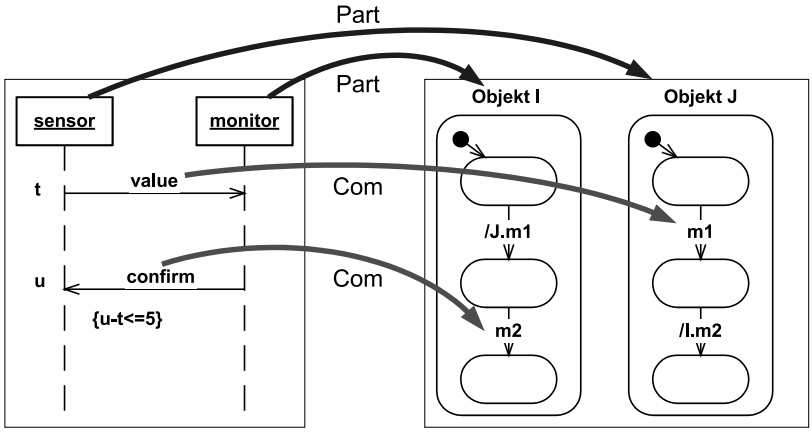


Abbildung 2.5: Systembezogene Anforderungsspezifikation

$M \mapsto E \cup Ch$ erweitert wird. Die Funktion *Part* bildet Instanzen auf Objekte ab. Die Funktion *Com* bildet Nachrichten auf Ereignisse oder Kanäle ab (Abb. 2.5). Diese Funktionen werden benötigt, um eine Anforderungsdefinition mit einem System zu verknüpfen. Ihr Gebrauch wird ausführlich im folgenden Kapitel beschrieben.

2.1.1.2 Semantik

Sequenzdiagramme werden vor allem dazu verwendet, eine zeitliche Ordnung auf Ereignissen zu spezifizieren. Zu jeder Nachricht gibt es ein Sende- und ein Empfangsereignis. Ein Ereignis ist eindeutig einer Instanz zugeordnet. Der UML-Standard verlässt sich weitgehend auf ein intuitives Verständnis von Sequenzdiagrammen. Eine spezielle formale Semantik von UML-Sequenzdiagrammen wird nicht spezifiziert. Daher ist es ein naheliegendes Vorgehen, auf der bewährten Semantik von Message Sequence Charts aufzusetzen [Ren98]. Dort wird die so genannte *visuelle Ordnung* von Nachrichten verwendet.

- Ereignisse sind entlang der Lebenslinie von Instanzen total geordnet.
- Das Versenden von Nachrichten erfolgt vor dem Empfang, d.h. ein Sendeereignis einer bestimmten Nachricht findet vor dem Empfangsereignis statt.

Hierdurch wird eine *partielle* Ordnung auf Ereignissen definiert. Über Instanzen hinweg werden Ereignisse nur durch Nachrichten geordnet. Fehlen zwischen den Instanzen Nachrichten, können derartig unabhängige Ereignisse beliebig angeordnet sein. Das führt dazu, dass im Diagramm optisch höher angeordnete Nachrichten, die möglicherweise intuitiv als früher stattfindend eingeschätzt werden, später ausgeführt werden. Daher wird als Erweiterung zum MSC Standard in der späteren Analyse die Möglichkeit eingeräumt, die totale Anordnung von Nachrichten in eine totale Anordnung von Ereignissen zu überführen. In diesem Fall verbieten sich überkreuzende Nachrichten. Diese Erweiterung ist jedoch *optional* und im Normalfall wird die visuelle Ordnung in ihrer ursprünglichen Form benutzt.

Schleifen können in eine Menge von einfachen Sequenzdiagrammen überführt werden, indem sie anhand ihrer erlaubten Iterationszahlen entfaltet werden. Ein Sequenzdiagramm mit Schleifen gilt dann als durchlaufen, wenn mindestens *ein* einfaches Sequenzdiagramm aus dieser Menge durchlaufen wird.

Neben asynchroner Kommunikation, wie bei MSCs, gibt es die Option, synchrone Kommunikation zu spezifizieren, d.h. das Sende- und Empfangsereignis fallen zeitlich zusammen. Allerdings ist in einer Spezifikation nur eine Form zulässig. Welche Kommunikationsart angemessen ist, hängt vom jeweiligen Anwendungsbereich ab.

2.2 Implementierungsnahe Verhaltensmodellierung

Nachdem im vorherigen Abschnitt beschrieben wurde, wie sich Anforderungen mit Hilfe von Sequenzdiagrammen so präzise beschreiben lassen, dass sie als Eingabe für einen automatisierten Verifikationsprozess geeignet sind, wird nun ein Gegenstück für eine formale Repräsentation einer Implementierung benötigt. Da Programmcode eine Verifikation mit Details überfrachten würde, ist eine abstraktere Darstellung erforderlich. Die UML bietet für eine diskrete Verhaltensmodellierung Zustandsdiagramme, die auf Harels Statecharts [HN96] basieren. Anders als diese Statecharts werden UML-Zustandsdiagramme im objektorientierten Kontext eingesetzt, d.h. sie be-

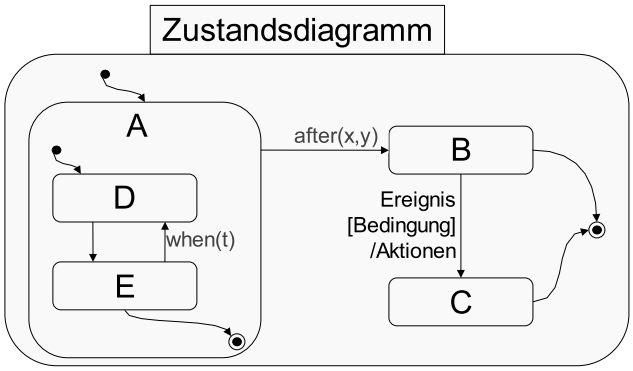


Abbildung 2.6: Zustandsdiagramm

schreiben das Verhalten von Objekten und Instanzen.[5] Dabei wird davon ausgegangen, dass ein Zustandsdiagramm einem bestimmten Objekt zugeordnet ist und dessen Verhalten beschreibt.

Viele reale Systeme schränken die Kommunikation zwischen Objekten ein. In QNX kommunizieren Prozesse über Kanäle. In objektorientierten Programmiersprachen werden Beziehungen zwischen Objekten mit Zeigerstrukturen oder Referenzen hergestellt. Diese Kommunikationsbeziehungen können entweder überall dort angenommen werden, wo sie sich implizit aus den dynamischen Modellen erschließen lassen. Beispielsweise impliziert das gerichtete Verschicken von Ereignissen eine solche Kommunikationsbeziehung. Oder Kommunikationsbeziehungen werden gesondert in einem Objektdiagramm dargestellt. Es wird in dieser Arbeit zunächst von der ersten Alternative mit Zustandsdiagrammen als alleinige Beschreibungssprache ausgegangen. Später wird diese Sichtweise um eine Beschreibung der Kommunikationsstruktur in Form eines Objektdiagramms ergänzt werden. Dieser zweite Ansatz aus Abschnitt 2.3 ist auf einen speziellen Anwendungskontext zugeschnitten und daher nicht mit einer Analyse von allgemeinen UML-Modellen kompatibel.

2.2.1 Zustandsdiagramme

Zustandsdiagramme beschreiben Software in diskreter Weise durch Zustände und Transitionen. Transitionen können mit *Ereignissen, Bedingungen* und

[5] In [HG97] wird von Harel selbst eine objektorientierte Variante für Statecharts vorgeschlagen.

Aktionen beschriftet sein. Ereignisse können im Moment ihrer Verarbeitung eine Transition aktivieren. In der UML ist nur maximal ein Ereignis pro Transition erlaubt.[6] Ob eine Transition schaltet und einen Zustandswechsel bewirkt, hängt von der Transitionsbedingung ab. Wird sie zu *true* ausgewertet, ist ein Zustandswechsel möglich. Schaltet eine Transition, werden ihre Aktionen ausgeführt. Beispielsweise werden Zuweisungen an Variablen vorgenommen, Funktionen aufgerufen oder neue Ereignisse verschickt.

Eine besondere Ausdruckskraft erhalten Zustandsdiagramme durch die hierarchische Untergliederung einer Verhaltensbeschreibung (Abb. 2.6). Dieses Merkmal ermöglicht erst eine praktische Anwendung bei nicht trivialen Systemen, da ohne diese Möglichkeit Modelle schnell unübersichtlich werden. Ein Zustand kann Unterzustände enthalten, mit denen sein Verhalten verfeinert wird. Weiterhin lassen sich orthogonale Regionen verwenden, um nebenläufiges Verhalten zu modellieren.

Ausgehend von diesem Grundmodell gibt es mittlerweile unzählige Varianten von Zustandsdiagrammen, die sich durch zulässige Syntax und realisierte Semantik unterscheiden. Die Variante von Zustandsdiagrammen, die im UML-Standard beschrieben wird, enthält Konstrukte, die für den Anwendungsbereich dieser Arbeit - der Modellierung von Echtzeitanwendungen - weniger relevant sind, wie z. B. Synchronisationszustände. Obwohl eine Berücksichtigung solcher Konstrukte während einer automatisierten Analyse oft durchaus möglich ist, kann dies zu einem beträchtlich höheren Analyseaufwand im Sinne des Zustandsraums führen und damit die Größe analysierbarer Modelle zum Teil drastisch einschränken. Daher werden in dieser Arbeit nur Elemente von UML-Zustandsdiagrammen zugelassen, die sich zur Modellierung von Echtzeitsystemen sinnvoll einsetzen lassen. Andererseits wird die Flexibilität zur Modellierung von zeitabhängigen Übergängen erhöht, indem eine gegenüber dem Standard leicht erweiterte Version des *after*-Konstrukts als Trigger für Transitionen zugelassen wird. Die wichtigsten Einschränkungen sind die folgenden:

- Es wird keine Nebenläufigkeit *innerhalb* von Zuständen zugelassen. Nebenläufigkeit wird in der UML durch orthogonale Regionen innerhalb eines Zustandes modelliert (*concurrent state*) und ist in einer frühen Phase der Spezifikation sinnvoll, um eine Überspezifikation zu vermeiden. Bei der implementierungsnahen Modellierung von Echtzeitsyste-

[6] Statecharts aus [HN95] erlauben mehrere Ereignisse, die verknüpft werden können. Im Kontext der UML-Semantik ist dies nicht sinnvoll.

men wird dieses Konstrukt üblicherweise nicht verwendet. Diese Einschränkung ist angemessen, weil hier Zustandsdiagramme verwendet werden, um das Verhalten von Threads zu modellieren, die kein nebenläufiges Verhalten aufweisen können. Da ein vollständiges Systemmodell aus einer Menge von kommunizierenden Zustandsdiagrammen besteht, ist Parallelität *zwischen* Threads modellierbar und erwünscht. Diese Argumentationsweise entspricht der aus [SGW94a]. Die dort beschriebene ROOM-Methode ist auf Echtzeitsysteme zugeschnitten und erlaubt ebenfalls keine nebenläufigen Unterzustände.

- Alle Ausdrücke, die an Transitionsbeschriftungen zugelassen sind (Bedingungen und Zuweisungen), sind auf eine einfache Form beschränkt. Diese Einschränkungen ergeben sich aus einer auf Timed Automata [BLL+95] basierenden Analyse. Prinzipiell ist laut UML-Standard jede Syntax für Bedingungen oder Zuweisungen zulässig. Um die in Kapitel 3 eingeführten Verfahren zur automatischen Analyse entscheidbar zu halten, muss auf nicht triviale Erweiterungen dieser Syntax verzichtet werden. Da Zustandsdiagramme nicht geeignet sind, komplexe algorithmische Probleme nachzubilden, stellt diese Einschränkung in der Praxis keine große Beschränkung da.

- Folgende Konstrukte werden nicht betrachtet: Zusammengesetzte Transitionen, History-Konnektoren und mit Zuständen verknüpfte Aktionen (Entry-, Exit- und Do-Aktionen). Obwohl eine entsprechende Erweiterung sich oft leicht konzipieren ließe, fielen diese Konstrukte den Bemühungen zum Opfer, einen vernünftigen Kompromiss zwischen Mächtigkeit der Sprache und Aufwand der Analyse im Sinne des Zustandsraums zu finden.

Im Folgenden werden diejenigen Elemente von Zustandsdiagrammen beschrieben, die in späteren Kapiteln weiter behandelt werden. Für eine vollständige Sprachbeschreibung von Zustandsdiagrammen wird auf den UML-Standard verwiesen [OMG01].

In dieser Arbeit bestehen Zustandsdiagramme aus Zuständen und Transitionen. Es werden einfache Zustände (*basic*), Endzustände (*final*) und zusammengesetzte Zustände (*composite*) unterschieden. Ein einfacher Zustand enthält keine Unterzustände. Ein zusammengesetzter Zustand hat wenigstens einen Unterzustand. Bei der Ausführung ist zu einem bestimmten Zeitpunkt

immer genau einer dieser Unterzustände aktiv . Mit Hilfe von zusammengesetzten und einfachen Zuständen wird eine Baumstruktur gebildet, wobei sich zusammengesetzte Zustände an den inneren Knoten befinden und einfache Zustände an den Blättern. Diese Hierarchie erlaubt eine schrittweise Verfeinerung des Verhaltens komplexer Systeme.

Jeder zusammengesetzte Zustand enthält genau einen Unterzustand, der als Startzustand gekennzeichnet ist. Er kann höchstens einen Endzustand haben. Wird ein zusammengesetzter Zustand durch eine Default-Transition betreten, aktiviert man zugleich den untergeordneten Startzustand. Wird der Endzustand eines zusammengesetzten Zustandes betreten, wird ein Ereignis (*completion event*) ausgelöst, welches die Abarbeitung dieser Hierarchieebene signalisiert. Dieses Ereignis kann nur von triggerlosen Transitionen (Transitionen ohne explizit bezeichnetes Ereignis als Trigger), die den übergeordneten Zustand verlassen, abgefangen werden. Da, wie oben erwähnt, nur einfache Transitionen betrachtet werden, hat jede Transition einen eindeutigen Quell- und Zielzustand. Diese Beschreibung führt zu folgender Definition.

Definition. Ein Zustandsdiagramm ist ein Tupel $< S, I, T, L, child, Var >$. $S = S_b \cup S_c \cup S_f$ ist die nicht leere Vereinigung der Mengen der einfachen Zustände S_b, der zusammengesetzten Zustände S_c und der Endzustände S_f. Der Wurzelzustand $root \in S_c$ ist der einzige Zustand, der keinen übergeordneten Zustand hat. $I \subseteq S_b \cup S_c$ ist die Menge der Startzustände, die in der graphischen Darstellung Ziel einer Default-Transition sind.

Eine Funktion *init* bildet jeden zusammengesetzten Zustand auf einen seiner Unterzustände ab, seinem Startzustand $s_0 \in I$. Die Funktion *child* bildet zusammengesetzte Zustände auf die Menge ihrer direkten Unterzustände ab. Die Menge aller Unterzustände erhält man durch $child^+$, dem irreflexiven transitiven Abschluss von *child*. Mit *child* wird eine Zustandshierarchie über S mit *root* als Wurzel induziert. Var enthält die Menge der Variablen eines Zustandsdiagramms.

T ist eine Menge von Transitionen. Eine Transition $t =< s_{src}, l, s_{tg} >$ ist ein Tupel, das einen Quellzustand $s_{src} \in S \setminus S_f$ und einen Zielzustand $s_{tg} \in S$ enthält. Die Transitionsbeschriftungen $l \in L$ sind von der Form $e[g]/a$. Dabei ist e ein Ereignis, das eine Transition triggert, g enthält eine Transitionsbedingung und a ist eine Liste von Aktionen, die ausgeführt werden, wenn die Transition schaltet. Alle Angaben der Transitionsbeschriftung können entfallen.

Das Triggerereignis e ist ein einfaches, explizit vermerktes Ereignis. Zulässig sind ebenfalls triggerlose Transitionen, bei denen ein explizites Ereignis e fehlt und die dann implizit durch ein Completion-Ereignis ausgelöst werden. Weiterhin sind zeitabhängige Ereignisse möglich. Wird an einer Transition als auslösendes Ereignis $after(min, max)$ benannt, bedeutet dies, dass nach $t \in [max, min]$ Zeiteinheiten nach Betreten des Quellzustands der Transition ein zeitabhängiges Ereignis ausgelöst wird.

In der UML gibt es ein Konstrukt $after(timeout)$, das genau nach $timeout$ Zeiteinheiten ein Zeitereignis auslöst, das zum Verlassen des Zustands führt. Die Motivation für die Erweiterung dieses Konzepts war, dass im Bereich von Echtzeitanwendungen oft mit Worst-Case-Execution-Times (WCET) gearbeitet wird. Exakte Ausführungszeiten lassen sich aus praktischen Gesichtspunkten selten garantieren. Meistens werden obere Grenzen für die Ausführungszeiten angegeben. Da Zustandsdiagramme zur Darstellung von algorithmischen Details wenig geeignet sind, wird mit der erweiterten Form des *after*-Konstrukts die Möglichkeit gegeben, den zeitlichen Effekt von Berechnungen zu modellieren.

Da Berechnungen einen Einfluss auf den Kontrollfluss haben können, weil Transitionsbedingungen vom Resultat dieser Berechnungen abhängig sein können, wird die Unvorhersagbarkeit eines Ergebnisses durch nichtdeterministische Verzweigungen ausgedrückt, da die tatsächlich durchführbaren Berechnungen auf der UML-Ebene auf Grundoperationen beschränkt bleiben müssen.[7]

Neben $after$-Transitionen wird noch ein zweiter zeitlicher Trigger unterstützt: Um auf eine globale Uhr zuzugreifen, kann folgender Ausdruck als Transitionstrigger verwendet werden: $when(x == c)$, wobei c eine Konstante aus der Menge der natürlichen Zahlen ist. Zum Beispiel lässt sich mit $when(x == 5)$ eine Transition 5 Zeiteinheiten nach Systemstart auslösen.

Wird kein Ereignis angegeben, so werden Transitionen durch ein implizites Completion-Ereignis $\surd$ getriggert. Für einen zusammengesetzten Zustand wird dieses Ereignis ausgelöst, wenn der untergeordnete Endzustand erreicht wird. Im Falle eines einfachen Zustandes wird ein Completion-Ereignis aktiviert, wenn der Zustand betreten wird. Dieses entspricht der intuitiven Vorstellung, dass eine triggerlose Transition in diesem Fall zum sofortigen Verlassen des Zustands führt, wenn die Transitionsbedingung erfüllt ist. Laut Stan-

[7] Die möglichen Operation werden in Kapitel 3 bei der Einführung von Zeitautomaten erläutert.

dard haben Completion-Ereignisse Priorität über andere Ereignisse. Dieses kann dadurch realisiert werden, dass eine Implementierung von Zustandsdiagrammen Completion-Ereignisse an die Spitze einer Ereigniswarteschlange setzt.

Eine *Transitionsbedingung* g ist ein boolescher Ausdruck und als *Aktion* ist das Versenden von neuen Ereignissen an beliebige Zustandsdiagramme oder die Wertezuweisung von Variablen (zum Beispiel $x := 1$) erlaubt. Die genaue Syntax von Zuweisungen wird von dem verwendeten Analysewerkzeug vorgegeben, das im Kapitel 3.1 eingeführt wird.

Eine vollständige Beschreibung eines Softwaresystems besteht aus einem *System* von kooperierenden Objekten mit Zustandsbeschreibung.

Definition. Ein System von Objekten ist ein Tupel $< sc = \{sc_{obj} | obj \in Obj\}, E, Var_{glob} >$. Die Menge sc enthält Zustandsdiagramme und die Menge E die Kommunikationsereignisse. Var_{glob} umfasst die Menge der globalen Variablen. Alle anderen Variablen sind als lokal anzusehen. Es wird vorausgesetzt, dass Zustände und Transitionen von verschiedenen Zustandsdiagrammen paarweise disjunkt sind. Ansonsten sind entsprechende Umbenennungen vorzunehmen.

2.2.1.1 Semantik von Zustandsdiagrammen.

In der Literatur gibt es eine Vielzahl von Vorarbeiten zur Definition einer formalen Semantik für Zustandsdiagramme. Prinzipiell lassen sich nahezu unbegrenzt viele Semantiken definieren, da es diverse semantische Interpretationsmöglichkeiten von Zustandsdiagrammen gibt. In [HN95] werden einige Entwurfsalternativen für eine Semantik aufgelistet und [vdB94] führt einen umfassenden Vergleich von semantischen Varianten durch. Laut [EW00] gehört zu den grundlegendsten Entscheidungen bei der Interpretation von Zustandsdiagrammen

- die Unterscheidung zwischen Anforderungs- oder Implementierungssemantik und
- die Unterscheidung zwischen einem objektorientierten Kontext oder einem Kontext der strukturierten Analyse.

Der Begriff Anforderungssemantik soll nicht missverstanden werden. Auch hier geht es um ein Modell der Implementierung, allerdings auf einer abstrakten Ebene, auf der Details der Realisierung noch keine Rolle spielen. Sowohl eine Anforderungs- als auch die Implementierungssemantik erklären eine Diagrammart, mit der eine Implementierung abstrakt dargestellt werden kann. Sie unterscheiden sich lediglich vom Abstraktionsniveau, das eingenommen wird.

Auf der Anforderungsebene werden idealisierte Annahmen gemacht. Da in einem frühen Entwurfsstadium ohnehin nicht bekannt ist, wie leistungsfähig die spätere Hardware ist und was für ein Zeitverhalten das verwendete Betriebssystem aufweist, wird zunächst eine perfekte Technologie unterstellt: Die Übergänge zwischen den Zuständen erfolgen so schnell, dass der Zeitverlust zu vernachlässigen ist und sie als zeitlos angesehen werden können. Ein Vorteil dieser Variante ist die oft effizientere Analysierbarkeit. Im Gegensatz dazu hat eine Implementierungssemantik die Realisierung in Form von ausführbaren Code im Blickfeld und kann daher derartige vereinfachende Annahmen nicht machen.

Die frühen „klassischen“ Definitionen von Zustandsdiagrammen gemäß Harel in [Har87, HN96] beziehen sich auf einen Einsatz innerhalb der strukturierten Analyse. Hier spielen Aktivitäten eine wichtige Rolle, die von Zustandsdiagrammen kontrolliert werden. Im objektorientierten Umfeld dagegen dienen Zustandsdiagramme vor allem der Modellierung von objektinternen Verhalten. Je nach Stimulus von außen und internem Zustand lässt sich eine Reaktion des Objektes präzise definieren. In diesem Sinne wurden Zustandsdiagramme in die UML integriert,[8] und damit zum ersten Mal einem Standard unterworfen.

Der UML-Standard enthält allerdings weder eine formale, noch wenigstens eine präzise Semantik, sondern gibt nur Richtlinien vor, die eine UML-konforme Formalisierung einhalten muss. Insbesondere enthält der Standard explizite „Semantic Variation Points“. Darunter werden semantische Alternativen verstanden, die für eine UML-konforme Semantik möglich sind. Die nicht formale Semantik der UML lässt einerseits ein breites Spektrum von Anwendungsbereichen zu. Andererseits stellt sie ein fast unüberwindliches Hindernis beim Austausch von Modellen zwischen verschiedenen Analysewerkzeugen dar.

[8] In der UML sind auch andere Verwendungen von Zustandsdiagrammen vorgesehen, z. B. zur Spezifizierung von Protokollen. Darauf wird hier nicht weiter eingegangen.

Im Standard werden folgende Anforderungen an eine hypothetische Maschine gestellt, welche die Semantik realisiert:

- Ereignisse werden in eine Struktur eingereiht, die einer Warteschlange entspricht. Die genaue Implementierung wird offen gelassen. Dabei ist auch Spielraum für Realisierungen gelassen, die bestimmten Ereignissen Priorität einräumen.

- Ein Dispatcher entnimmt das aktuelle Ereignis und sorgt dafür, dass in Abhängigkeit von der gegenwärtigen Konfiguration (aktive Zustände, Variablenbelegung, usw.) ein Wechsel in eine neue Konfiguration stattfindet.

Semantik eines Systems von Zustandsdiagrammen. Innerhalb des UML-Standards bleibt die Frage offen, auf welche Weise ein System von Zustandsdiagrammen zusammenarbeitet. Das hängt damit zusammen, dass für verschiedene Anwendungsbereiche und Abstraktionsebenen unterschiedliche Semantiken optimal sind. In einem Kontext arbeiten die Prozesse, welche die Zustandsdiagramme implementieren, echt parallel, da sie verteilt auf verschiedenen Prozessoren realisiert werden. Steht nur ein Prozessor zur Verfügung, ist eine echt parallele Bearbeitung nicht möglich. Andererseits kann dies auf einer höheren Abstraktionsebene eine plausible Vereinfachung sein.

In vielen Semantiken, wie beispielsweise der von Statemate [HN96], wird die Frage der Kommunikation zwischen Zustandsdiagrammen nicht behandelt. Notwendig wird dies erst, wenn man sich im objektorientiertem Kontext bewegt. So wird in [HG97] von Harel eine objektorientierte Erweiterung von Zustandsdiagrammen beschrieben. Neben ereignisbasierter Kommunikation werden auch blockierende Funktionsaufrufe an andere Komponenten betrachtet.

Im Folgenden werden exemplarisch einige Arbeiten zur Semantikdefinition von Zustandsdiagrammen aufgelistet.

- Harel definiert in [HN96] eine Semantik für die von ihm entwickelten Zustandsdiagramme [HG97]. Diese Semantik ist auf die strukturierte Analyse zugeschnitten und kann daher für die UML nicht ohne Anpassungen übernommen werden. Sie wird im Werkzeug Statemate realisiert. Allerdings ist diese Semantik nur „präzise“, nicht aber formal definiert. In [DJHP97] wird eine formale Beschreibung durchgeführt.

- Harel entwirft ebenfalls eine objektorientierte Semantik-Variante [HG97]. Eine Implementierung liegt im kommerziellen Werkzeug Rhapsody vor, deren Einzelheiten in [HK04] beschrieben werden. In [HK99] werden neben der Vererbung von Zustandsdiagrammen vor allem die Kollaboration einer Menge von Zustandsmaschinen betrachtet. Es werden die asynchrone Kommunikation über Ereignisse und die synchrone Kommunikation mit Methodenaufrufen, einschließlich Verklemmungen aufgrund zyklischer Abhängigkeiten, untersucht. In dieser Semantik ist der interne Aufbau der kommunizierenden Zustandsdiagramme sehr einfach gehalten und zeitliche Aspekte werden nicht untersucht.

- In [LMM99] werden hierarchische Automaten benutzt, um eine formale Semantik für ein Zustandsdiagramm zu definieren. Allerdings wird das Zusammenspiel mehrerer Zustandsdiagramme nicht untersucht.

- [LPP99] konzentriert sich ebenfalls auf ein Zustandsdiagramm und lässt Kommunikationsaspekte weitgehend außer Acht. Die Semantik wird in Form von Pseudo-Code beschrieben. Sie liegt näher an einer Implementierungssicht, es wird keine zeitlose Verarbeitung vorausgesetzt und Ereignisse werden in Warteschlangen verwaltet. Zeitaspekte spielen nur am Rande eine Rolle.

- In [EW00] wird eine Anforderungssemantik für UML-Zustandsdiagramme erarbeitet. Die Geschwindigkeit, mit der Modelle ausgeführt werden, ist zu vernachlässigen. Insbesondere sind Transitionen zeitlos. Die Semantikdefinition orientiert sich an [DJHP97], überträgt aber die Konzepte auf einen objektorientierten Kontext. Sie berücksichtigt synchrone und asynchrone Kommunikation, Objekterzeugung und -terminierung, ein synchrones und asynchrones Zeitmodell, jedoch keine Ereigniswarteschlangen, sondern Ereignismengen.

Innerhalb dieser Arbeit wurde Wert darauf gelegt, dass die verwendete Semantik kompatibel mit einem konkreten Anwendungsbereich ist. Ziel ist eine Analyse von Modellen, die Robotersteuerungen beschreiben. Für diesen Bereich sind Echtzeitaspekte besonders wichtig, die daher auch angemessen in der Semantik berücksichtigt werden müssen. Insbesondere sollten sich Intervalle für die Ausführungszeiten einzelner Aktionen angeben lassen. Weiterhin muss die Semantik die Kommunikation innerhalb eines Systems von

Zustandsdiagrammen unterstützen, die jeweils das Verhalten nebenläufiger Prozesse beschreiben.

Die hier verwendete Semantik ist durch [EW00] motiviert. Wie oben erwähnt, wird dort sowohl eine zeitsynchrone Semantik (Ereignisse werden nur mit dem Auftreten eines Taktsignals verarbeitet), als auch eine zeitasynchrone Semantik (Ereignisse werden verarbeitet, sobald sie auftreten) beschrieben. Die erste Variante ist vor allem sinnvoll, wenn Hardware modelliert wird. Da in dieser Arbeit ausschließlich Software-Systeme betrachtet werden, in denen ein Signal unabhängig von einem Taskzyklus[9] verarbeitet wird, ist für uns vor allem der zweite Ansatz relevant.

In [EW00] wird zwischen synchroner und asynchroner Kommunikation unterschieden. Im ersten Fall wird der auslösende Prozess solange blockiert, bis die Anfrage abgearbeitet wurde. Andernfalls kann er sofort nach Absetzen eines Ereignisses weiterarbeiten. Wir konzentrieren uns hier auf den zweiten Fall, da die Kommunikation über Ereignisse die wichtigste Kommunikationsart innerhalb von UML-Zustandsdiagrammen darstellt. Das auf Anforderungsebene synchrone Kommunikation entbehrlich ist, sei z. B. mit dem Erfolg von Statemate belegt, das ebenfalls keine synchrone Kommunikation unterstützt.

Um Konflikte beim Zugriff auf Variablen zu verhindern, erlaubt die Semantik aus [EW00] während eines Schritts nur lesenden Zugriff auf alle Variablen. Änderungen werden innerhalb eines Schrittes auf eine Variablenkopie angewendet (*Double Buffering*). Erst wenn der Schritt abgeschlossen ist, werden die geänderten Werte übernommen.

Eine Verdoppelung aller Variablen würde den Zustandsraum bei einer Verifikation negativ beeinflussen. Da keine nebenläufigen Regionen zugelassen sind und so immer maximal eine Transition schaltet, entfällt die Notwendigkeit von Double Buffering, da keine Konflikte beim Zugriff auf Variablen auftreten können. Auch dieser Performanzvorteil spricht für den Verzicht auf nebenläufige Regionen.

Dieses gilt nur für lokale Variablen. Wird auf eine Variable global von mehreren Threads aus zugegriffen, ist dies Teil der modellierten Spezifikation und der Anwender trägt die Verantwortung dafür, so zu modellieren, dass ungewollte Konflikte nicht auftreten.

[9] Auf anderen Abstraktionsebenen spielen Zyklen für den Echtzeitbereich eine wichtige Rolle. Allerdings ist z. B. der Takt eines Kommunikationsprotokolls ggf. selbst Gegenstand des Modells, nicht aber der Semantik.

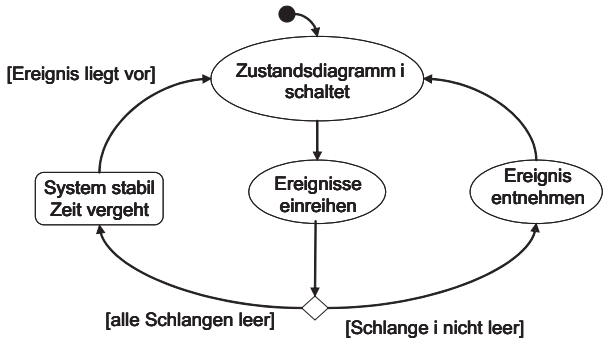


Abbildung 2.7: Semantik eines Systems von Zustandsdiagrammen

Ein wichtiges Prinzip innerhalb von objektorientierten Zustandsdiagrammen ist *Run-to-Completion*: Ein neues Ereignis wird erst verarbeitet, wenn alle vorausgegangenen Ereignisse komplett verarbeitet wurden. In [EW00] ist dieses erfüllt, da auch die Ereignismengen jedes Zustandes dupliziert werden. Neue Ereignisse werden daher erst nach einem Schritt wirksam. Aber auch ohne Double Buffering ist *Run-to-Completion* in Abwesenheit synchroner Ereignisse gewährleistet, wenn die Berechnung der Menge der schaltenden Transitionen abgeschlossen ist, bevor neue Ereignisse auftreten können, so wie es in dieser Arbeit der Fall ist.

Informelle Semantikbeschreibung. Wir beschreiben zunächst den groben Ablauf innerhalb eines Systems von Zustandsdiagrammen. Später gehen wir detaillierter auf den Ablauf innerhalb einer Zustandsmaschine ein. Es wird dabei von folgenden Annahmen ausgegangen (siehe Abb. 2.7):

- Zeit vergeht nur, wenn alle Ereigniswarteschlangen leer sind. Das System befindet sich in einem *stabilen* Zustand.
- Wenn ein zeitlich getriggertes Ereignis eintritt, wird das System *instabil*. Durch das Ereignis können Aktionen ausgelöst werden, die weitere Ereignisse auslösen können. Diese Ereignisse können an beliebige Zustandsdiagramme (auch an das eigene) gerichtet sein. Sie werden hinten in die Ereigniswarteschlange eingefügt und ebenfalls abgearbeitet. Im instabilen Zustand vergeht keine Zeit.
- Sollten in dieser Phase mehrere Zustandsdiagramme Ereignisse in ihrer Warteschlange aufweisen, wird ein Zustandsdiagramm nichtdeterminis-

tisch ausgewählt, um das erste Ereignis der ausgewählten Warteschlange zu behandeln. Dieser Nichtdeterminismus wird als vom Modellierer bewusst spezifiziert angenommen.

- Sind alle Ereigniswarteschlangen aller Zustandsdiagramme leer, kehrt das System in den stabilen Zustand zurück. Dieser Zustand kann nur durch zeitlich getriggerte Ereignisse verlassen werden. Spontane Ereignisse einer nicht modellierten Umgebung werden nicht behandelt.

Unterschiede zur Semantik von [EW00]:

- Es werden hier Ereigniswarteschlangen statt Ereignismengen verwendet. Dies ist näher am Standard, ohne dass vom Konzept einer Anforderungssemantik abgerückt werden muss. Completion-Ereignisse erhalten dabei gegenüber anderen Ereignissen eine höhere Priorität. Ansonsten gibt es keine Prioritätsabstufungen zwischen Ereignissen.

- Completion-Ereignisse werden auch bei einfachen Zuständen schon mit deren Betreten ausgelöst, um von ihnen abzweigende, triggerlose Transitionen zu ermöglichen.

- Es wird kein Double Buffering verwendet. Dieser Mechanismus vergrößert den Zustandsraum mit der Verdopplung aller Variablen, löst aber nicht das Problem von Nichtdeterminismus bei mehrfachen Schreibzugriffen.

- Es werden hier weder ein synchrones Zeitmodell, noch die dynamische Objektbehandlung berücksichtigt. Im anvisierten Anwendungsbereich der Robotersteuerung werden in aller Regel weder Steuerungskomponenten erst zur Laufzeit aktiviert/deaktiviert, noch kann die Verarbeitung von Ereignissen auf diskrete Zeitpunkte beschränkt werden. Sollte die dynamische Objektbehandlung zur Modellierung einer fortschrittlichen Steuerung nötig sein, ließe sich die Lösung, alle Strukturen von Beginn an anzulegen und deren Aktivierungszustand über entsprechende boolesche Variablen zu verwalten, leicht umsetzen.

Ablauf in einem Zustandsdiagramm. Zu Beginn werden der Zustand *root* und danach rekursiv alle untergeordneten Startzustände als aktiv gekennzeichnet, d.h die Menge $init^*(root)$ bildet die Startkonfiguration des Zustandsdiagramms. Eine Transition gilt als *aktiviert*, wenn der Quellzustand

aktiv ist, das auslösende Ereignis e der Ereigniswarteschlange zur Bearbeitung entnommen wird und die Transitionsbedingung erfüllt ist. Da möglicherweise mehrere Transitionen aktiviert werden, die den gleichen Quellzustand verlassen, können Konflikte auftreten. Dieses kann ebenfalls auftreten, wenn die aktiven Zustände hierarchisch ineinander geschachtelt sind. Da in dieser Arbeit keine Diagramme mit nebenläufigen Regionen existieren, sind mehrere aktive Transitionen in einem Zustandsdiagramm immer im Konflikt. Die UML-Semantik sieht vor, dass Zustände, die weiter unten in der Hierarchie stehen, Priorität haben. Wenn Transitionen von demselben Zustand ausgehen, wird nichtdeterministisch ausgewählt.

Wenn eine Transition schaltet, werden die in a enthaltenen Aktionen ausgeführt. Zunächst werden dann alle Zustände als inaktiv markiert. Der Zielzustand s_{tg} und alle seine Oberzustände, bis hin zum Wurzelzustand, werden aktiv. Wenn s_{tg} kein einfacher Zustand ist, wird der Startzustand, der ihm untergeordnet ist, ebenfalls aktiv. Dies gibt auch für alle weiteren untergeordneten Startzustände, d.h. alle Zustände $init^*(s_{tg})$ werden aktiv.

Im UML-Standard wird als Datenstruktur für die Verarbeitung von Ereignissen eine Warteschlange favorisiert. Die genaue Realisierung wird aber nicht weiter spezifiziert, so dass auch beispielsweise Ereignisse mit Prioritäten zulässig wären. Solange die Ereigniswarteschlange gefüllt ist, werden Ereignisse nach und nach abgearbeitet. Dabei wird ein neues Ereignis erst zur Verarbeitung freigegeben, wenn die Verarbeitung des vorherigen Ereignisses vollständig abgeschlossen ist (*Run-to-Completion*).

Abschließend wird die bisherige informelle Semantikbeschreibung durch einige formale Definitionen präzisiert. Diese Semantik orientiert sich an [EW00] und übernimmt dort verwendete Definitionen, soweit sie im Kontext dieser Arbeit sinnvoll sind. An bestimmten, oben beschriebenen Stellen wird von der dort definierten Semantik abgewichen.

Die Grundlage der Semantikdefinition ist eine Kripke-Struktur mit Zeiterweiterung (Clock Labelled Kripke Structure, im Folgenden kurz CLKS). Sie ist ein Tupel $(Var, Act, \longrightarrow, ci, \sigma_0)$, die aus folgenden Komponenten besteht: Var enthält eine Menge von Variablen und Act eine Menge von Aktionen. Var umfasst modellspezifische Variablen, die aus dem Zustandsdiagramm übernommen werden, aber auch Standardvariablen, deren Bedeutung sich aus der Konstruktion des Transitionssystems ergibt. Dazu gehören die Variable cfg, welche die Menge der aktiven Zustände des Zustandsdiagramms beinhaltet, und die Variable inp, welche eine Warteschlange von Ereignissen

enthält.

Die Relation $\rightarrow$ definiert Übergänge zwischen Wertzuweisungen der Variablen, die mit Aktionen aus Act beschriftet sind. Mit $\sigma \xrightarrow{A} \sigma'$ wird ein Übergang von der Wertezuweisung σ zur Wertezuweisung σ' beschrieben, der die Aktionen $A \subseteq Act$ auslöst. Diese Beschriftung A kann die Aktionen der Transition, die durch ihr Schalten den Übergang bewirkt hat, enthalten. Zulässig ist auch die Beschriftung mit einer Zeitspanne δ, wenn es sich um einen zeitlichen Schritt handelt.

Startzustand des Transitionssystems ist die Wertezuweisung σ_0. $\sigma_0(cfg)$ enthält alle Zustände, die sich durch Verfolgung der Default-Transitionen ergeben. Außerdem ist $first(\sigma_0(inp)) = \emptyset$, d. h. die Ereigniswarteschlange ist zu Beginn leer, und alle Uhren sind auf 0 initialisiert.

Wertezuweisungen σ können mit der Funktion ci Invarianten zugewiesen werden. Für jede Transition, die von einem Zustand aus cfg abgeht und eine zeitliche Bedingung $c \geq n$ aufweist, wird ein $c \leq n$ der Invariante hinzugefügt. Die Invariante ergibt sich aus der Konjunktion der Bestandteile.

Wir definieren zunächst eine lokale Semantik bezogen auf ein Zustandsdiagramm und setzen daraus später eine Gesamtsemantik zusammen.

2.2.1.2 Definition lokale Semantik eines Zustandsdiagramms

Die Übergangsrelation $\longrightarrow$ des CLKS setzt sich aus den Relationen $\longrightarrow_{time}$, $\longrightarrow_{change}$ und $\longrightarrow_{superstep}$ zusammen [EW00]. Der Übergang $\longrightarrow_{time}$ modelliert das Verstreichen von Zeit. Mit $\longrightarrow_{change}$ tritt eine Änderung der Eingangsvariablen auf - entweder aufgrund eines Umgebungsereignisses oder eines Timeouts. Dieses führt zu einem $\longrightarrow_{superstep}$, der Reaktionen auf die Änderungen umsetzt. Die Übergänge $\longrightarrow_{change}$ und $\longrightarrow_{superstep}$ erfolgen dabei gemäß einer Anforderungssemantik zeitlos. Ein Superstep ist wiederum in Einzelschritte $\longrightarrow_{step}$ unterteilt. Somit ergeben sich folgende Sequenzen:

$$\xrightarrow{A} = \xrightarrow[time]{\delta} ; \longrightarrow_{change}; \xrightarrow[superstep]{A}$$

$$\xrightarrow[superstep]{A} = \xrightarrow[step]{A_1} \cdots \xrightarrow[step]{A_n} .$$

Dabei gilt $A = A_1 \cup \cdots \cup A_n$. Außerdem enden die Einzelschritte in einem stabilen Zustand mit $first(\sigma_0(inp)) = \emptyset$.

Zeitlicher Schritt. Alle Uhren $cl \in Clk$ werden um ein δ erhöht, wenn für alle Zwischenwerte ϵ die Invarianten ci gültig sind, d.h. die Invarianten werden zu keinem Zeitpunkt im Verlauf der Zeit verletzt:

$$\sigma \xrightarrow[time]{\delta} \sigma' \Leftrightarrow \sigma' = \sigma[cl/\sigma(cl) + \delta] \wedge \forall \epsilon \in [0, \delta] : ci_{\sigma''}$$

$$mit\ \sigma'' = \sigma[cl/\sigma(cl) + \epsilon].$$

Nicht zeitliche Veränderung. Möglich sind neue Ereignisse, spontane Änderungen von Umgebungsvariablen und Timeouts:

$$\longrightarrow_{change} = \longrightarrow_{event} \cup \longrightarrow_{value} \cup \longrightarrow_{timeout} .$$

Spontanes Ereignis. Es wird ein neues Ereignis E aufgenommen. Anders als in [EW00] wird hier angenommen, dass Ereignisse in einer Warteschlange verwaltet werden:

$$\sigma \longrightarrow_{event} \sigma' \Leftrightarrow \sigma' = \sigma[inp/insert(\sigma(inp), E)].$$

.

Spontane Wertänderungen. Werteänderungen lassen die aktuelle Konfiguration cfg und die Ereigniswarteschlange unverändert. Derartige Werteänderungen können von anderen Komponenten verursacht werden.

$$\sigma \longrightarrow_{value} \sigma' \Leftrightarrow \sigma \neq \sigma' \wedge \sigma(cfg) = \sigma'(cfg) \wedge \sigma(inp) = \sigma'(inp)$$

.

Timeout. Bei einem Timeout ist kein zeitlicher Schritt mehr möglich, da z. B. eine zeitgetriggerte Transition schalten muss.

$$\sigma \longrightarrow_{timeout} \sigma' \Leftrightarrow \sigma = \sigma' \wedge \nexists \sigma''\ \sigma \longrightarrow_{time} \sigma''.$$

Schritt. Es wird eine Transition $T*$ ausgewählt, die schaltet. Dabei wird auch die Transitionsbedingung g berücksichtigt. Da wir keine orthogonalen Zustände betrachten, kann jeweils nur eine einzelne Transition schalten, nämlich genau eine derjenigen Transitionen mit der höchsten Priorität in der gegebenen Konfiguration. Sollte keine Transition schalten können, ist $T* = \emptyset$.

Die Funktion $nextconfig$ berechnet die neue Konfiguration nach dem Transitionsübergang $T*$. Dazu wird der Quellzustand der ausgewählten Transition und alle davon hierarchisch abhängigen Zustände aus der Konfiguration entfernt und der Zielzustand mit allen umschließenden Zuständen und den per Default-Transition aktivierten Unterzuständen hinzugefügt:

$$\sigma \xrightarrow[step]{A} \sigma' \Leftrightarrow \neg empty(\sigma(inp_{id})) \wedge A = act(T*)$$

$$\wedge \sigma' = \sigma''[cfg/nextconfig(\sigma(cfg), T*),$$

$$inp/insert(compl, pop(\sigma(inp_{id}))),$$

$$\bigwedge_{x \in \sigma'(cfg) \setminus \sigma(cfg)} tmr_x/0].$$

σ'' ergibt sich dabei aus der Auswertung der Aktionen:

$$\sigma'' = eval_\sigma(act(T*)).$$

Die Variablen tmr_x sind Uhren, welche die Verweildauer innerhalb eines Zustandes beschreiben. Sie werden auf 0 zurückgesetzt.

Weiterhin ist *compl* ein Completion-Ereignis, das ausgelöst wird. Wechselt ein Zustandsdiagramm in eine Konfiguration, in der ein Completion-Ereignis auftritt, wird dieses in *inp* eingefügt. Wird ein einfacher Zustand s betreten, wird dessen Completion-Ereignis $\surd_s$ erzeugt. Wird ein Endzustand $final_s$ betreten, wird das Completion-Ereignis des übergeordneten Zustandes $\surd_s$ generiert:

$$\forall s \in cfg \wedge s \notin cfg' \Rightarrow \surd_s = first(\sigma(inp))$$

$$wenn\ type(s) = BASIC\ \vee (type(s) = OR \wedge final_s \in cfg)\}.$$

Die Variable cfg bezeichnet dabei die aktuelle Konfiguration und cfg' die Vorgängerkonfiguration. Anders als bei [EW00] löst hier auch das Betreten eines einfachen Zustands ein entsprechendes Completion-Ereignis aus. Damit sind auch triggerlose Transitionen möglich, die von einfachen Zustände wegführen. Weitere Ereignisse können durch Auswerten der Aktionen mit *eval* erzeugt werden.

2.2.1.3 System von Zustandsdiagrammen

Dem Ansatz von [EW00] folgend, werden Kennungen $id \in OID$ für jedes Objekt mit einem enthaltenen Zustandsdiagramm eingeführt. Die Kennungen werden verwendet, um die diagrammspezifischen Variablen zu indizieren. Ein Objekt id besitzt damit eine Konfiguration cfg_{id}, eine Ereigniswarteschlange inp_{id}, lokale Uhren $tmr_{s,id}$ und lokale Variablen v_{id}. An der Schrittdefinition ändert sich im wesentlichen nur, dass zwischen unterschiedlichen Objekten id unterschieden wird:

$$\sigma \xrightarrow[step]{A_{id}} \sigma' \Leftrightarrow \neg empty(\sigma(inp_{id})) \wedge A_{id} = act(T_{id}*)$$

$$\wedge \sigma' = \sigma''[cfg_{id}/nextconfig(\sigma(cfg_{id}), T_{id}*), inp_{id}/insert(compl, pop(\sigma(inp_{id}))),$$

$$\bigwedge_{x \in \sigma'(cfg_{id}) \setminus \sigma(cfg_{id})} tmr_{x,id}/0].$$

Die Wertezuweisung σ'' ergibt sich wie oben aus der Auswertung der Transitionsaktionen. Um Ereignisse an andere Zustandsdiagramme zu verschicken, wird die Funktion $eval$ wie folgt erweitert:

$$eval_{\sigma}(send\, id.signal) = \sigma[inp_{id}]/insert(signal, inp_{id})]$$

Die Definition eines zeitlichen Schrittes $\xrightarrow[time]{\delta}$ ändert sich kaum, außer dass jetzt die Uhren *aller* Zustandsdiagramme einbezogen werden müssen.

Die Definition von $\longrightarrow_{change}$ ist in einem System von Zustandsdiagrammen nicht mehr notwendig, da Änderungen jetzt nicht mehr spontan auftreten, sondern von anderen Zustandsdiagrammen bewirkt werden. Das Auftreten von spontanen Ereignissen wird nicht mehr berücksichtigt, da wir für die Verifikation ein geschlossenes System brauchen. Alle Ereignisse aus der Umwelt sollen dabei Teil des Modells werden.

Ein Superstep gilt als abgeschlossen, wenn eine stabile Konfiguration vorliegt, d.h. alle Ereigniswarteschlangen sind leer:

$\sigma \xrightarrow[superstep]{A} \sigma' \Leftrightarrow \sigma \xrightarrow[step]{A_1} \ldots \xrightarrow[step]{A_n} \sigma' \wedge \forall id \in OID : first(\sigma'(inp)) = \emptyset$ mit $A = A_1 \cup \cdots \cup A_n$.

Damit ist die Beschreibung der hier verwendeten Sequenz- und Zustandsdiagramme abgeschlossen. Bevor darauf aufbauend ein automatisches Analyseverfahren entwickelt wird, wird das Thema Modellierung in eine weitere

Richtung vertieft. Danach wird die Dynamik-Analyse in Kapitel 3 ausführlich behandelt.

2.3 Modellierung von speziellen Anwendungsgebieten

Bisher bezog sich die Modellierung auf allgemeine UML-Modelle. Werden UML-Modelle für einen bestimmten Anwendungsbereich erstellt, stellt sich oft heraus, dass es spezifische Konstrukte gibt, die dort immer wieder auftauchen und die sich schlecht mit dem Kern der UML erfassen lassen. Dieses Kapitel widmet sich der anwendungsspezifischen Anpassung von UML. Es stellt da, wie die Entwicklung eines Profils helfen kann, prägnante Modelle zu erstellen und wie damit die Voraussetzung für eine automatische Analyse geschaffen wird, die später beschrieben wird.

Es wird hier die Erstellung eines Prozessmodells für das Echtzeitbetriebssystem QNX untersucht. Dabei wird ein statisches Architekturmodell eingeführt, das die Verbindung von Prozessen über Kanäle darstellt. Das interne Verhalten von Threads wird durch Zustandsdiagramme modelliert. Im folgenden Abschnitt werden die QNX-spezifischen Konstrukte beschrieben, die bei einer Modellierung berücksichtigt werden sollen. Danach wird darauf aufbauend ein UML-Profil definiert, das eine solche Modellierung ermöglicht.

2.3.1 Das Echtzeitbetriebssystem QNX

QNX ist ein Echtzeitbetriebssystem, dessen Threads feste Prioritäten haben und vollständig unterbrechbar sind [Krt99]. Es ist ein Microkernel-Betriebssystem, das aus einem kleinen Kernbereich, welcher die minimal erforderliche Menge von Diensten für den Betrieb des Systems zur Verfügung stellt, und einer Menge von kooperierenden Prozessen besteht. Jeder Prozess enthält wenigstens einen ausführbaren Thread (maximal sind 32767 Threads für einen Prozess möglich). Sowohl Prozesse als auch Threads besitzen eindeutige Identifier und können dynamisch erzeugt und beendet werden. Threads werden durch *ThreadCreate*() erzeugt.

Für die Kommunikation zwischen Threads stellt QNX insbesondere Message-Passing zur Verfügung. Message-Passing ist eine synchrone, blockierende Kommunikationsform. Ein sendender und ein empfangender Thread werden

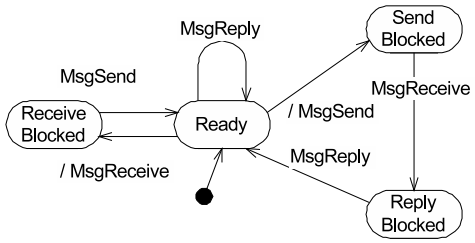


Abbildung 2.8: Zustände eines Threads beim Message-Passing

solange blockiert, bis beide Kommunikationspartner zum Informationsaustausch bereit sind. Dann wird eine Nachricht direkt von einem Adressraum in den anderen kopiert. Die Blockierung des Senders wird durch eine bestätigende Antwort aufgehoben.

Die Kommunikation wird mit bestimmten Anweisungen eingeleitet. Ein Client-Thread, der eine Nachricht an einen Server versenden möchte, setzt ein $MsgSend()$ ab, um die Kommunikation einzuleiten. Er blockiert solange, bis der Empfänger ein $MsgReceive()$ sendet, die Nachricht verarbeitet und mit einem abschließenden $MsgReply()$ die Blockierung endgültig aufhebt. Der Client-Thread durchläuft dabei einem *send*- und *reply*-blockierten Zustand (Abb. 2.8).

Wenn ein Server-Thread ein $MsgReceive()$ sendet, blockiert dieser ebenfalls solange, bis ein Client zur Synchronisation bereit ist und ein $MsgSend()$ generiert. Der Kommunikationsvorgang wird wie oben durch ein $MsgReply()$ des Servers beendet.

Message-Passing findet nicht direkt von Thread zu Thread, sondern über Kanäle statt. Ein Kanal wird von einem Server-Thread mit $ChannelCreate()$ erzeugt. Client-Threads, die diesem Server eine Nachricht senden wollen, verbinden sich mit $ConnectAttach()$ mit diesem Kanal.

Kanäle ordnen Servern Nachrichten unter Erhaltung der Reihenfolge zu. Jeder Kanal enthält drei Warteschlangen, in denen Threads abhängig von ihrem Blockiert-Zustand einsortiert werden (Abb. 2.9).

- $Receive - Queue$: Diese enthält Server-Threads, die auf eine Nachricht warten.

- $Send - Queue$: In dieser Warteschlange werden Client-Threads gespeichert, die eine Nachricht gesendet haben, die aber noch nicht empfangen

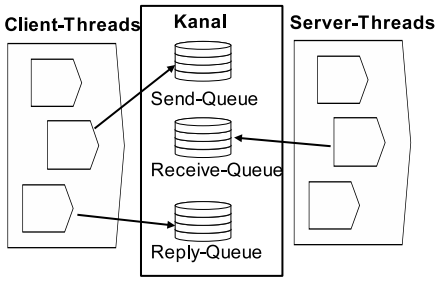


Abbildung 2.9: Warteschlangen eines Kommunikationskanals

wurde.

- $Reply - Queue$: Diese Warteschlange enthält Client-Threads, die eine Nachricht gesendet haben, die empfangen, aber noch nicht beantwortet wurde.

Folgende Befehle finden unter QNX beim Message-Passing Anwendung: Zunächst sendet ein Client den Befehl $sts = MsgSend(coid, smsg, rmsg)$. Dabei bezeichnet sts eine Statusmeldung der Antwort (ok oder Fehler), $coid$ den Kanal, $smsg$ die Anfrage des Clients und $rmsg$ einen Zeiger auf die spätere Antwort des Servers. Ein Server leitet die Kommunikation durch $rcvid = MsgReceive(chid, rmsg, msg\_info)$ ein: Er erwartet eine Nachricht vom Kanal $chid$, die unter der Adresse $rmsg$ gespeichert wird. msg_info enthält Informationen über den Client und $rcvid$ ist eine Kennung der Anfrage, die für die Rücksendung benötigt wird. Der Kanal ordnet Client und Server einander zu. Die Kommunikation wird vom Server mit $MsgReply(rcvid, sts, smsg)$ beendet. Die Antwort wird über die Anfragenkennung $rcvid$ zugeordnet und die Server-Antwort $smsg$ für den Client in seinen Adressraum unter $rmsg$ kopiert.

Neben dem synchronen, blockierenden Message-Passing gibt es in QNX einen Kommunikationsmechanismus mit nicht blockierenden Nachrichten, den so genannten Pulsen. Diese werden eher im Sinne eines UML-Ereignisse als zur Nachrichtenübertragung benutzt. Hierfür stehen die Anweisungen $MsgSendPulse()$ und $MsgReceivePulse()$ zur Verfügung. Sie verfügen über die gleichen Parameter wie die einfachen $MsgSend()$- und $MsgReceive()$-Befehle. Pulse werden auch durch ein $MsgReceive()$ empfangen. Da Pulse nicht beantwortet werden, bleibt die Antwortkennung $rcvid$ unbelegt. Im Unterschied zum Message-Passing werden Pulse asynchron verschickt, d.h. der

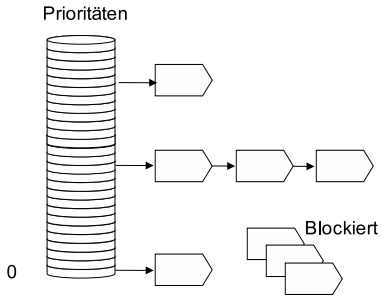


Abbildung 2.10: Prioritäten-Warteschlangen

Client-Thread blockiert nicht. Anders als UML-Ereignisse wird ein Puls in der Send-Queue des Kanals solange zwischengespeichert, bis ein Thread ein $MsgReceive()$ oder $MsgReceivePulse()$ sendet.

Jedem Thread ist für das Scheduling eine Priorität zugeordnet, die in QNX zwischen 1 und 63 liegen kann. Anhand dieser Priorität trifft der Scheduler eine Entscheidung, welchem Thread Rechenzeit zugebilligt wird. Der Scheduler wählt immer denjenigen Thread aus, der die höchste Priorität hat. Threads gleicher Priorität werden in eine Warteschlange ihrer Prioritätsstufe einsortiert (Abb. 2.10).

Die Ausführung des Threads, der gerade den Prozessor nutzt, wird zeitweise unterbrochen, sobald ein Kernel-Call wie zum Beispiel ein $MsgSend()$ aufgerufen wird. Ändert sich der Ausführungszustand eines Threads, wird eine neue Scheduling-Entscheidung getroffen. Ein Wechsel des auszuführenden Threads findet in folgenden Fällen statt:

- Der laufende Thread blockiert, da er auf das Auftreten eines Ereignisses warten muss (zum Beispiel auf die Antwort eines Servers bei Message-Passing). Der Thread wird dann aus der Prioritätenwarteschlange entfernt. Sobald der Thread nicht mehr blockiert ist, wird er erneut an das Ende der Warteschlange seiner Priorität angefügt.

- Ein Thread mit höherer Priorität wird in die Prioritäten-Warteschlange eingefügt. Dieses tritt zum Beispiel auf, wenn nach einer Kommunikation die Blockierung des Threads wieder aufgehoben wird.

- Der laufende Thread gibt durch die Anweisung $sched\_yield()$ den Prozessor frei. Der Thread wird an das Ende der Warteschlange seiner Priorität verschoben.

Beim Message-Passing ist noch eine Besonderheit zu beachten. Damit Client-Anfragen an einen Server mit der richtigen Priorität bearbeitet werden, wird die Client-Priorität an den Server vererbt. Dieses beugt dem Problem der *Priority Inversion* vor [Krt99]. Nachdem die Anfrage beantwortet wurde, wird der Server auf seine ursprüngliche Priorität zurückgesetzt.

2.3.2 Modellierung von QNX-Anwendungen

Um eine möglichst weitreichende Analyse zu ermöglichen, wird eine explizite Präzisierung der Modellierung benötigt. In der UML sind dafür Stereotypen vorgesehen. Diese Stereotypen werden wie Marken verwendet, deren Anwendung auf UML-Elemente zu einer Spezialisierung ihrer Bedeutung führt. Eine Menge solcher Stereotypen für einen bestimmten Anwendungsbereich kann zu einem UML-Profil zusammengefasst werden. Im Folgenden wird ein Profil zur Beschreibung von Kommunikationsstrukturen in QNX eingeführt und als *Profil für QNX-Anwendungen* bezeichnet. Bei der Auswahl der Konstrukte, die in das Profil aufgenommen werden, sind vor allem die Kriterien ausreichende Ausdrucksstärke, Analysierbarkeit und intuitive Verwendbarkeit maßgeblich.

UML-Profil für QNX-Anwendungen. Die UML ist geeignet, eine Vielzahl von Systemen zu modellieren. Natürlich stößt auch eine relativ mächtige Sprache angesichts der Komplexität realer Systeme an ihre Grenzen, zumal wenn der Sprachumfang überschaubar gehalten werden soll. Für spezielle Anwendungsgebiete werden derartige Lücken mit Hilfe von so genannten Profilen gefüllt. Beispielsweise sind die Möglichkeiten zur Modellierung von Echtzeitsystemen im UML-Kern begrenzt. Mit dem Profil für *Schedulability, Performance and Time* [Obj05] wurde ein erster Versuch von der OMG unternommen, dieses wichtige Feld für die UML zu erschließen. Allerdings bleibt das Profil unter den Möglichkeiten, die beispielsweise die ROOM-Methode [SGW94a] bietet, so dass es als erster Schritt verstanden werden sollte. Für eine spezifischere Modellierung ist eine weitere Verfeinerung notwendig, die sich an den Charakteristika des Anwendungsbereiches orientiert. Dieses ist insbesondere nötig, wenn das Modell Ausgangspunkt für eine automatisierte Verifikation ist. In diesem Abschnitt wird exemplarisch ein Profil entwickelt, das Sprachelemente aus QNX darstellt. In Kapitel 3 wird dargestellt, wie darauf aufsetzend automatisch verifiziert werden kann.

Kennzeichnend für UML-Profile ist, dass sie den UML-Kern im wesentlichen unangetastet lassen. D.h. Änderungen am UML-Metamodell werden möglichst konservativ betrieben, da es die Grundlage für die Implementierung von UML-Werkzeugen liefert. Werden in einem Profil am UML-Metamodell Änderungen vorgenommen, ist die Verwendung von Standardwerkzeugen praktisch ausgeschlossen: Wird beispielsweise eine zweite Art eines Default-Konnektors für Zustandsdiagramme eingeführt, lässt sich das mit dem bisherigen UML-Metamodell nicht vereinbaren und alle aktuellen Werkzeuge wären erst nach einer Erweiterung seitens der Hersteller einsetzbar. Daher sollten sich UML-Profile auf die standardmäßigen Erweiterungsmechanismen der UML beschränken. Es handelt sich dabei um

Stereotypen (engl. Stereotypes): Sie können auf beliebige Elemente des UML-Metamodells angewendet werden. Sie geben dann diesem Modellelement eine neue, spezielle Bedeutung.

Eigenschaftswerte (engl. Tagged Values): Sie können für ein bestimmtes Stereotyp festgelegt werden und enthalten spezifische Werte. Beispielsweise kann für Tasks festgelegt werden, dass sie Ausführungszeiten enthalten.

Einschränkungen (engl. Constraints): Können die Verwendung von Stereotypen beliebig einschränken. Es kann z.B. definiert werden, dass in einem konsistenten Modell die Ausführungszeit eines Task kleiner sein muss, als dessen Deadline.

Viele UML-Werkzeuge unterstützen diese Erweiterungsmechanismen durch die freie Einführung beliebiger Stereotypen und damit verbundener Eigenschaftswerte und Einschränkungen. Daher lassen sich oft Profile, die sich auf diese Mechanismen beschränken, direkt mit Standardsoftware umsetzen.

Ziel der Modellierung im Zusammenhang dieser Arbeit ist eine Darstellung von Prozessen, Tasks und deren Kommunikationsbeziehungen (Applikationsstruktur) und eine Abstraktion des reaktiven Verhaltens von Softwarekomponenten. Die statische Architektur einer Anwendung kann mit Verteilungs- bzw. Komponentendiagrammen sowie eingebetteten Objektdiagrammen dargestellt werden. Für die Beschreibung des dynamischen Verhaltens der einzelnen Threads bieten sich Zustandsdiagramme an. Die Darstellung von Echtzeitaspekten wird dabei durch eine verfeinerte semantische Spezifikation der entsprechenden Objekte mittels einzuführender Stereotypen und

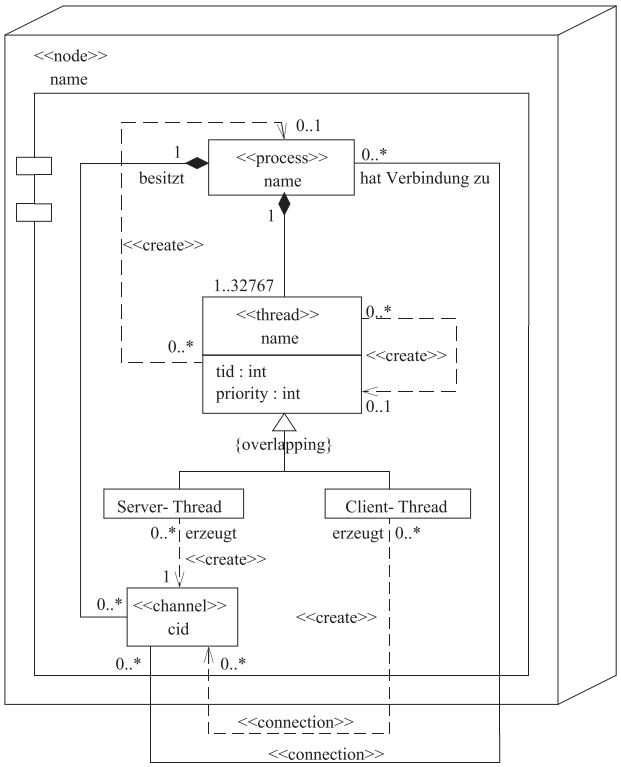


Abbildung 2.11: Metamodell für die Prozessmodellierung

Eigenschaftswerten realisiert. Mit den hier eingeführten Stereotypen ist jeweils ein QNX-Konstrukt verbunden, das die Grundlage der semantischen Interpretation darstellt. D.h. ein Stereotyp sollte so interpretiert werden, wie sich sein Gegenstück im QNX-Kontext verhält.

Prozesse und Kommunikationsstruktur. In standardisierten Profilen der OMG wird der Gebrauch von Stereotypen durch die Angabe einer abstrakten Syntax eingeschränkt. Eine solche Syntax wird als Klassendiagramm (Metamodell) dargestellt.

Die *Prozesssicht,* wie sie hier definiert ist, enthält Prozesse, die Threads enthalten und auf einem Hardware-Knoten laufen. Prozesse benutzen zur Kommunikation Kanäle. In Abbildung 2.11 ist ein Metamodell dazu angegeben.

Das Stereotyp << *node* >> bezeichnet Hardware-Knoten. Sie stellen physisch unabhängige Rechnerressourcen dar, die einen eigenen Prozessor

zur Verarbeitung von Prozessen bzw. Threads und einen eigenen Scheduler besitzen.

Jeder Prozess ($<<process>>$) enthält mindestens einen Thread. Maximal kann jeder Prozess gemäß den Vorgaben von QNX aus 32767 Threads bestehen. Zur Kommunikation werden einem Prozess Kanäle (Channels) und Verbindungen (Connections) zugeordnet.

Jeder Thread $<<$ *thread* $>>$ hat einen Namen, eine global eindeutige Kennnummer und einen Prioritätswert für das Scheduling. Bei der Kommunikation können Threads eine Sender- oder eine Empfängerrolle einnehmen (auch beides gleichzeitig), wenn sie als Client oder Server agieren. Server-Threads sind in der Lage, eine beliebige Anzahl von Kanälen $<<$ *channel* $>>$ zu erzeugen. Sie werden ebenfalls eindeutig durch eine Kennnummer identifiziert. Kanäle können zur Kommunikation genutzt werden, sobald andere Threads eine Verbindung $<<$ *connection* $>>$ mit ihnen eingegangen sind. Es sind auch Kommunikationsverbindungen zwischen Threads möglich, die Hardware-Grenzen überschreiten. Kanäle und Verbindungen werden von Threads erzeugt, aber den übergeordneten Prozessen zugeordnet. Die Erzeugung von Kanälen und Verbindungen, sowie von Prozessen und Threads, kann mit Hilfe von UML-Abhängigkeiten, die mit dem Stereotyp $<<$ *create* $>>$ versehen sind, dargestellt werden. Der Startprozess eines Knotens und der Main-Thread eines Prozesses darf nicht so gekennzeichnet werden, da sie automatisch erzeugt werden.

Dynamisches Verhalten der Threads. Das dynamische Verhalten von Threads wird durch Zustandsdiagramme dargestellt. Es wird angenommen, dass eine Folge von internen Aktionen unsichtbar innerhalb eines Zustandes stattfindet, solange sie keinen Einfluss auf das Kommunikationsverhalten hat. Aktivitäten, die umfangreiche Berechnungen enthalten, werden auf ihren zeitlichen Effekt reduziert dargestellt[10]. Alle anderen, die Kommunikation beeinflussenden Anweisungen, werden an Zustandsübergängen notiert. Dieses sind im Wesentlichen Kommunikationsbefehle, Anweisungen zur Erzeugung von Objekten oder zur Veränderung von Variablenwerten, sowie Transitionsbedingungen. Schleifen können durch Zyklen modelliert werden.

[10] In [Fir04] wird beschrieben, wie sich Slicing-Techniken zur Abstraktion einsetzen lassen. Denkbar wäre die Anwendung dieser Technik, um aus Quellprogrammen automatisch diese Darstellung zu gewinnen. Berücksichtigt werden müssten insbesondere Variablen, die in Transitionsbedingungen verwendet werden.

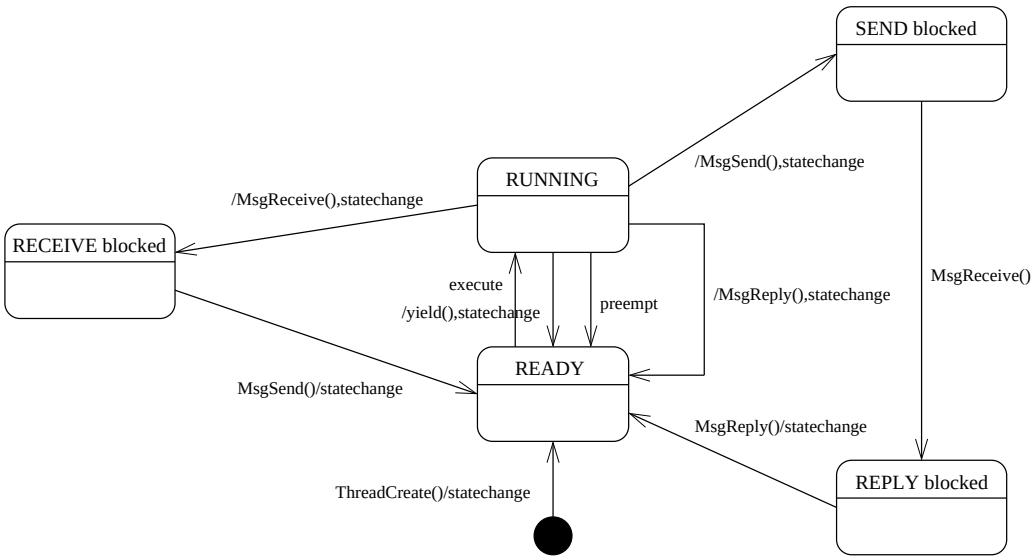


Abbildung 2.12: Zustände eines Threads

Zusätzlich müssen implizit alle Zustände betrachtet werden, in denen sich ein Thread bei der Kommunikation, beim Scheduling und bei der Prozessornutzung befinden kann. Es ist nicht notwendig, dass ein Benutzer dafür Zustandsdiagramme anlegt, da sie automatisch ergänzt werden. Abbildung 2.12 zeigt das Threadmodell aus QNX [Krt99] als Zustandsdiagramm. Ein Thread konsumiert Prozessorzeit, wenn er sich im Zustand *Running* befindet. Nur in diesem Zustand kann eine Kommunikation initiiert werden. Daher beginnen die Kommunikationszyklen für Clients und Server im Zustand *Running*. In Folge einer Kommunikation durch Message-Passing wird ein Thread blockiert. Nach einem Nachrichtenaustausch wird ein Thread erneut in die Warteschlange der ausführbaren Tasks einsortiert und ein Scheduling ausgelöst.

Jeder Wechsel des Ausführungszustandes eines Threads (*Ready*, *Running* und *Blocked)* bewirkt eine neue Scheduling-Entscheidung. Daher wird an jedem entsprechenden Zustandsübergang eine *statechange*-Aktion ausgelöst. Der Scheduler unterbricht dann den gerade laufenden Thread mit der Aktion *preempt* und wählt den nächsten Thread nach dem Scheduling-Verfahren aus. Dessen Ausführung wird durch *execute* gestartet. Wenn ein Thread explizit den Prozessor durch die Anweisung *yield*() freigibt, muss ebenfalls eine neue Scheduling-Entscheidung durch *statechange* herbeigeführt werden. Dieses geschieht ebenso bei der Erzeugung eines neuen Threads (Aufruf *ThreadCreate*()). Alle oben genannten Aktionen bzw. Ereignisse sind Calls bzw. Call Events. Sie triggern synchron Transitionen in internen, nicht modellierten Zustandsdiagrammen (Scheduler und Channels) oder in anderen Thread-Diagrammen. Dabei werden auch Parameter übergeben.

Der Anwender erstellt eine Abstraktion des Quellprogramms, indem im Wesentlichen Kommunikationsaktionen des Message-Passings und Aktivitäten mit ihrem Zeitverbrauch dargestellt werden. Bei der Transformation in ein ausführbares System von Zeitautomaten für die Verifikation wird dann (automatisiert) eine Ergänzung der benötigten Zwischenzustände für die Kommunikation stattfinden.

Um die zeitliche Wirkung einer Aktivität zu modellieren, wird der Eigenschaftswert *exectime* für die Ausführungszeit von Anweisungen eingeführt. Diese Ausführungszeit wird als *Worst-Case-Execution-Time* (WCET) interpretiert. Für eine präzisere Modellierung lassen sich ebenfalls minimale und maximale Grenzen der Ausführungszeit mit *minExectime* und *maxExectime* angegeben.

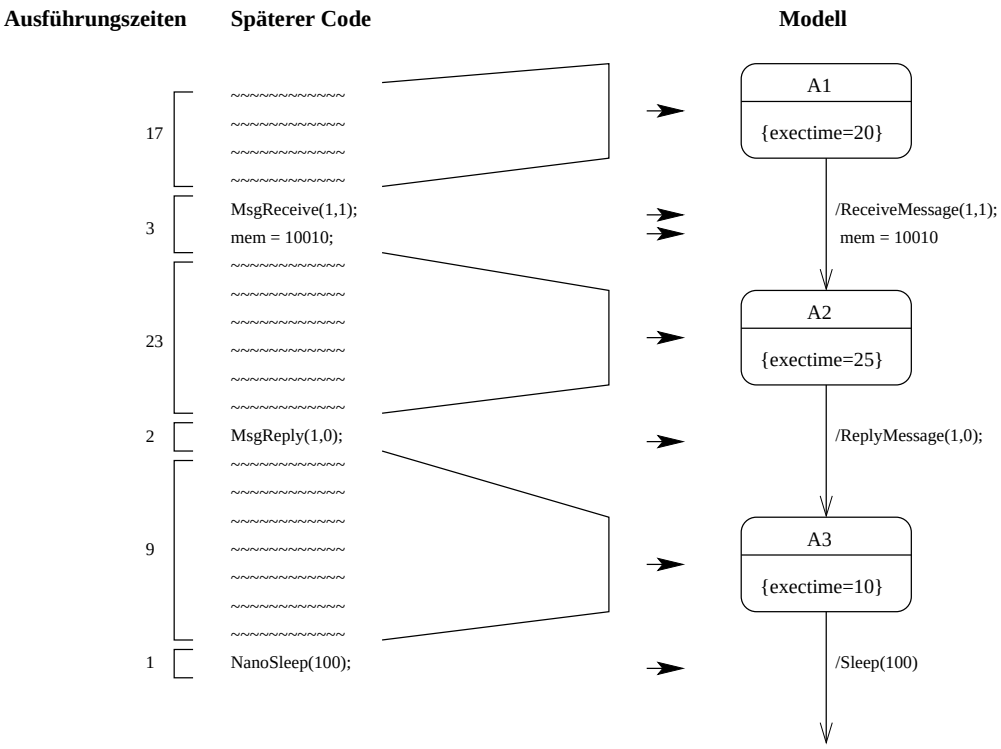


Abbildung 2.13: Beispiel für ein abstraktes Programm

Kommunikationsaktionen werden an Zustandsübergängen eingetragen. Die Laufzeiten, die ein bestimmtes Betriebssystem zur Initiierung der Kommunikation benötigt, sind beim Zeitverbrauch der vorgelagerten Aktivitäten zu berücksichtigen, soweit sie ins Gewicht fallen. In Abbildung 2.13 ist auf der linken Seite dargestellt, wie eine aus dem Quellprogramm gewonnene Abstraktion aussieht.

Folgende Aktionen ermöglichen die prägnante Modellierung von Kommunikation auf UML-Ebene, d.h. es handelt sich um Beschriftungen, die in Zustandsdiagrammen verwendet werden:

$SendMessage(chid, data)$ Diese Aktion führt zum kompletten Durchlauf eines Sendezyklus von einem Client-Thread ($MsgSend() - MsgReceive() - MsgReply()$). Der Parameter *chid* enthält die Kennnummer eines Kanals. Hier und im Folgenden steht *data* jeweils für Nutzdaten, die ver-

sendet werden. Dabei ist auch die Angabe eines geteilten Speicherbereichs statt konkreter Daten zulässig.

ReceiveMessage(*chid*) Diese Aktion versetzt einen Server-Thread in einen Wartezustand für Anfragen. Der Parameter *chid* wird wie oben verwendet.

ReplyMessage(*chid*, *sts*, *data*) Mit der Aktion *ReplyMessage*() beantwortet ein Server eine Anfrage eines Clients und hebt dessen Blockierung auf. Die Statusvariable *sts* enthält einen Wert für einen Fehlerstatus. Die Angabe *data* ist optional.

Yield() Dieser Befehl entspricht der *QNX*-Anweisung *schedYield*() eines Threads, mit der Rechnerressourcen zurückgegeben werden können.

SendPulse(*chid*) Die Anweisung entspricht dem Befehl *MsgSendPulse*() zum Versenden eines Pulses. Pulse dienen als Trigger für Aktionen des Kommunikationspartners. Als Parameter wird die Kanalkennnummer *chid* angegeben.

ReceivePulse(*chid*) Dieses entspricht dem *QNX*-Befehl *MsgReceivePulse*() zum Empfangen eines Pulses über einen Kanal mit der Kennnummer *chid*.

Sleep(*sec*) Der Befehl *Sleep*() modelliert die *Nanosleep*()-Anweisung aus *QNX*. Hiermit blockiert der Thread für *sec* Zeiteinheiten.

CreateProcess($name_p$) Diese Anweisung dient dem dynamischen Starten eines anderen Prozesses auf dem gleichen Hardware-Knoten. Dieser Prozess ist durch $name_P$ identifizierbar.

CreateThread(*tid*) Dieser Befehl startet einen anderen Thread des gleichen Prozesses. Durch *tid* ist der Thread eindeutig identifiziert.

CreateChannel(*chid*) Die Anweisung *CreateChannel*() erzeugt einen Channel mit der Kennnummer *chid*, um einen Nachrichtenempfang zu ermöglichen.

AttachConnection(*chid*) Mit *AttachConnection*() wird eine Verbindung zu einem Channel *chid* aufgebaut.

Weiterhin sind Zuweisungen an den Transitionsübergängen zulässig. Diese Zuweisungen an Variablen können dazu benutzt werden, um einen Schleifenzähler zu realisieren oder um Shared-Memory-Daten zu modellieren. Die genaue Syntax möglicher Zuweisungen ist von der gewählten Analysemethode abhängig. In diesem Fall werden nur Ausdrücke unterstützt, die vom Verifikationswerkzeug direkt analysierbar sind. Daher wird die genaue Syntaxbeschreibung auf Kapitel 3 verschoben. Im Wesentlichen sind einfache Zuweisungen an ganzzahlige und boolesche Variablen zulässig. In ähnlicher Weise lassen sich diese Variablen in Bedingungen an Transitionen verwenden.

Wie in der UML üblich, wird davon ausgegangen, dass jedes Zustandsdiagramm über einen besonders markierten Startzustand verfügt. Falls ein Thread terminiert, kann dem Diagramm ein Endzustand hinzugefügt werden. Dieser Zustand darf dann keine ausgehenden Transitionen besitzen. Je Zustandsübergang darf nur eine Kommunikationsaktion notiert werden. Diese kann mit Bedingungen oder Variablenmanipulationen kombiniert werden.

Definition von Tasks. Bei Echtzeitanwendungen soll verifiziert werden, ob Tasks ihre Deadlines einhalten. Dazu ist es notwendig, innerhalb einer Definition eines Threads Anfang und Ende eines bestimmten Tasks zu definieren. Dies ist insbesondere dann notwendig, wenn innerhalb eines Threads mehrere Tasks implementiert werden. Um diese Zuordnung für den Anwender möglichst flexibel zu halten, können beliebige Anweisungen mit den beiden Stereotypen $<< trigger >>$ und $<< response >>$ definiert werden. Diese kennzeichnen Beginn und Ende eines Task und werden an Transitionen notiert. Eine Verknüpfung von Taskaktivierung und -ende findet über den Eigenschaftswert *taskName* statt. Mit Hilfe des Eigenschaftswertes *deadline* können zeitliche Anforderungen in das Modell integriert werden. Abbildung 2.14 verdeutlicht dieses Konzept. Zur Vereinfachung fehlen in diesem Beispiel die üblichen Transitionsbeschriftungen. Es werden dort zwei Tasks definiert: Der rechte Zweig beschreibt einen regulären Ablauf und der linke die Behandlung einer Notfallsituation.

2.3.3 Beispiel

Die Anwendung des oben beschriebenen Profils wird an einem Beispiel verdeutlicht, das durch den Sonderforschungsbereich 562 motiviert ist. Ziel des SFB ist die Weiterentwicklung von Robotern mit parallelen Kinematiken, d.h. Strukturen, in denen die Antriebe selbst nicht bewegt werden, sondern

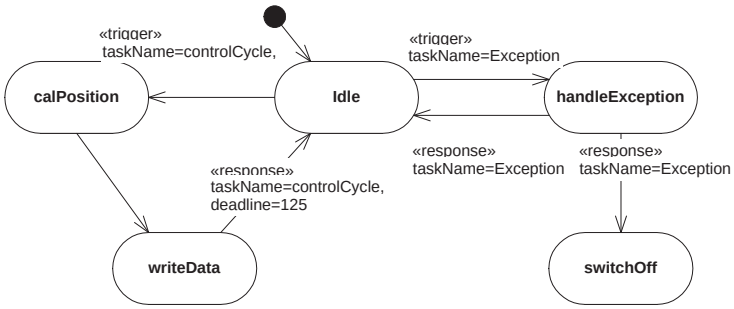


Abbildung 2.14: Definition von Tasks durch Trigger und Response

Abbildung 2.15: Prototypischer Versuchsträger eines Parallelroboters

fest mit einer Basis verbunden sind. Ein prototypischer Versuchsträger wird in Abbildung 2.15 gezeigt.

Die Steuerung besteht aus den vier Prozessen Programm, Bahnplanung, Regelung und Monitor. Ein zusätzlicher Prozess wickelt die Kommunikation über Firewire ab. Alle Prozesse enthalten einen Thread *main*, dem jeweils eine Priorität zugeordnet ist. Die Prozesse kommunizieren über Kanäle. Die statische Anwendungsarchitektur ist in Abbildung 2.16 dargestellt.

Das Verhalten eines Threads wird als Zustandsdiagramm modelliert (Abb. 2.17 und 2.18). Der Programm-Thread besitzt eine Liste von hintereinander auszuführenden Bewegungen. Ein Programm sendet Aktionsprimitive an die Bahnplanung. Aktionsprimitive werden abstrakt als Daten in Nachrichten verschickt. Mit der Kommunikationsaktion $SendMessage()$ wird eine Nach-

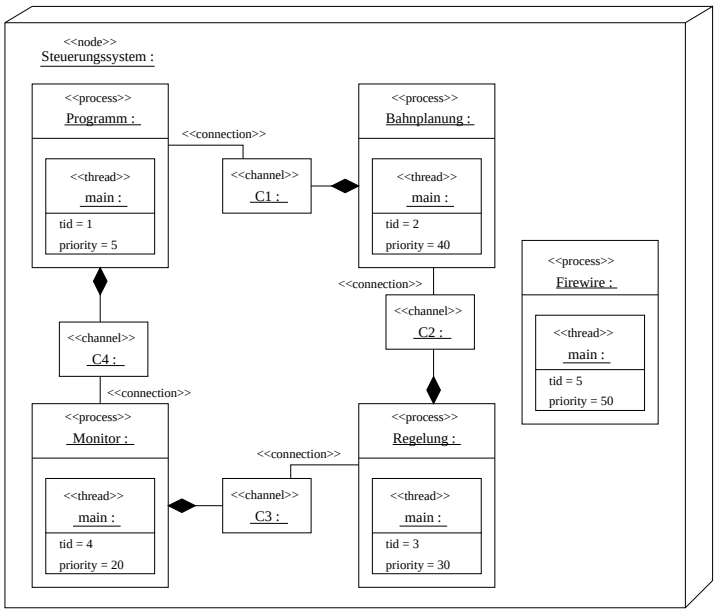


Abbildung 2.16: Architektur der Robotersteuerung

richt an den Kanal $C1$ verschickt. Aus diesem Kanal liest die Bahnplanung das aktuelle Aktionsprimitiv. Mit $ReceiveMessage()$ und $ReplyMessage()$ wird die Kommunikation zum Programm-Thread abgeschlossen.

In der Bahnplanung wird ein zeitintensiver Berechnungsvorgang gestartet. In dieser Zeit befindet sich die Kontrolle im Zustand $B3$. Anhand der Istdaten, modelliert durch die globale Variable *ddtdata* (Shared Memory), und den empfangenen Daten werden feinere Bewegungsdaten (Solldaten) für den Roboter berechnet (Operation hier vereinfacht als Addition dargestellt) und über den Kanal $C2$ an die Regelung geschickt.

Alle 125 Zeiteinheiten werden über die Firewire-Schnittstelle die Istdaten aktualisiert. Der Takt wird dabei mit einer $Sleep()$-Anweisung modelliert. Außerdem übermittelt der Thread die Solldaten aus dem Shared Memory an den Roboter. Die Regelung empfängt neue Solldaten und startet mit einen Puls den Monitor, der die Roboterbewegung überwacht. Hat der Monitor seine Aufgabe beendet, aktiviert er seinerseits mit einem Puls den Programm-Thread.

In diesem Beispiel soll das Einhalten der Deadline des gesamten Programmzyklus verifiziert werden. Vom Beginn des ersten Aktionsprimitivs im

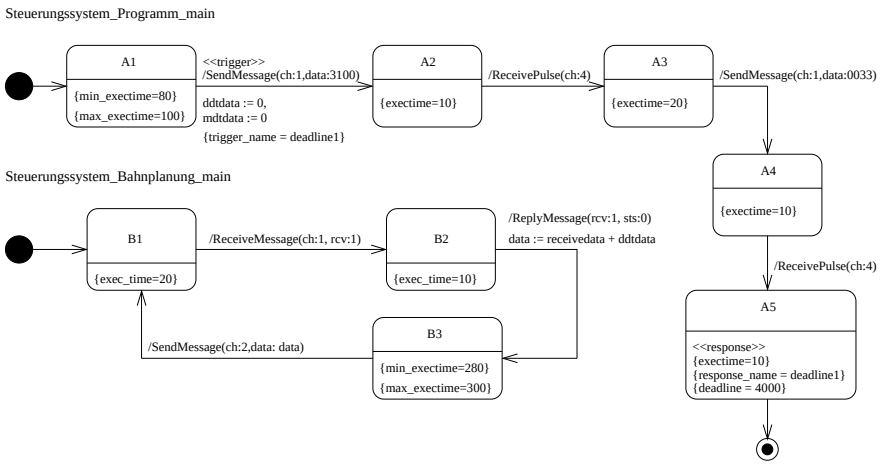


Abbildung 2.17: Verhaltensmodellierung des Programm-Threads und der Bahnplanung

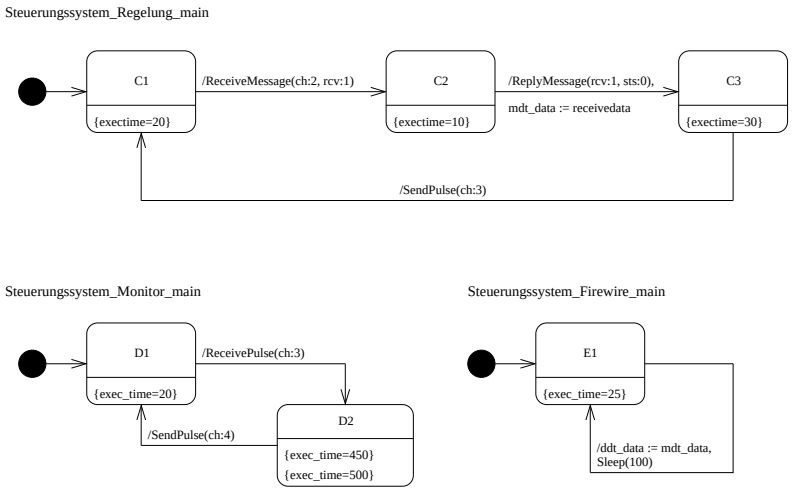


Abbildung 2.18: Verhaltensmodell von Regelung, Monitor, Firewire

Programm (Trigger) bis zum Ende des zweiten (Response) dürfen in diesem Beispiel maximal 4000 Zeiteinheiten vergehen. Die Verifikation kann automatisch mit dem in Abschnitt 3.4 beschriebenen Verfahren durchgeführt werden.

Kapitel 3

Dynamische Analyse

Die formale Analyse von Echtzeitsystemen kann auf vielfältige Weise vorgenommen werden. In der Literatur werden zahlreiche Formalismen beschrieben, die aufgrund ihrer Berücksichtigung von Zeit prinzipiell geeignet sind. Einen sehr guten Überblick über den aktuellen Stand der Forschung gibt [Wan04]. Besonders verbreitet zur Modellierung von Echtzeitsystemen sind Zeitautomaten. Zeitautomaten erlauben eine intuitive Modellierung mit einem Netzwerk von Prozessen, deren internes Verhalten durch eine Erweiterung von endlichen Automaten beschrieben wird. Zeitvariablen und davon abhängige Invarianten und Transitionsbedingungen stellen den Zeitbezug her. Synchronisationsaktionen und globale Variablen erlauben die Verknüpfung von Prozessen. Grenzen bei der Modellierung großer Systeme werden u. a. durch fehlende Konstrukte zur hierarchischen Untergliederung gesetzt.

Ein Formalismus kann nur als Grundlage für die Analyse von dynamischen UML-Modellen dienen, wenn er in einem Werkzeug implementiert wurde. In [Wan04] werden Werkzeuge aufgelistet, mit denen Echtzeitsysteme formal analysiert werden können. Die Leistungsfähigkeit dieser Werkzeuge ist generell schwer zu bewerten, da der Aufwand bei einer Verifikation stark von der Analyseaufgabe abhängt. Spezialisierte Datenstrukturen zur effizienten Speicherung des Zustandsraums liefern für einige Beispiele sehr gute Ergebnisse, können sich aber für andere Aufgaben als völlig ungeeignet erweisen.

In dieser Arbeit wird der Model-Checker UPPAAL benutzt. UPPAAL zeichnet sich gegenüber anderen Model-Checkern für Zeitautomaten nicht nur durch besonders effiziente Algorithmen aus. Als vorteilhaft hat sich auch eine durchgängige Weiterentwicklung des Werkzeugs erwiesen, die auch noch in jüngster Zeit Performancevorteile erreichen konnte. Die graphische Ober-

fläche hat diesem Werkzeug eine vergleichsweise große Nutzergemeinde verschafft. Insbesondere Studenten wird dadurch der Einstieg erleichtert. Auch wenn diese Arbeit unabhängig von einer graphischen Benutzeroberfläche für Zeitautomaten ist, profitiert sie indirekt davon, da die große Nutzergemeinde schnell Fehler des Werkzeugs aufdeckt und zur Stabilität von UPPAAL beiträgt.

In diesem Kapitel werden zunächst Zeitautomaten vorstellt, mit denen dank Werkzeugunterstützung Echtzeitsysteme analysiert werden können. Danach wird dargestellt, wie sich dieser Formalismus im Rahmen der UML nutzen lässt.

3.1 Zeitautomaten als formale Notation

Zeitautomaten (*timed automata*) sind eine Erweiterung von endlichen Automaten. Sie sind eine formale Notation zur Modellierung und Verifikation von Echtzeitsystemen. Aufbauend auf den grundlegenden Arbeiten von Alur und Dill [AD94] wurden mehrere Model-Checker implementiert, die Zeitautomaten als Eingabesprachen nutzen [Yov97, LPY97]. Diese Werkzeuge sind maßgeblich für die praktische Anwendung von Zeitautomaten.

Im ursprünglichen Ansatz von Alur und Dill sind Zeitautomaten als endliche Büchi-Automaten konzipiert worden, die um Variablen erweitert wurden, deren reelle Werte Zeit repräsentieren. Zeitvariablen können in Bedingungen an den Transitionen verwendet werden und erlauben so zeitabhängige Zustandswechsel. In einer Variante, den *Timed Safety Automaten*, werden Invarianten an Zuständen eingeführt, mit denen auf intuitive Weise ein Übergang in einen anderen Zustand erzwungen werden kann. Timed Safety Automaten sind in Werkzeugen wie UPPAAL [LPY97] und Kronos [Yov97] implementiert. Im Folgenden beziehen wir uns auf diese Variante.

3.1.1 Zeitautomaten

Zeitautomaten sind eine Erweiterung von endlichen Automaten. Sie bestehen aus beschrifteten Zuständen und Transitionen. In den Beschriftungen kann Bezug auf Zeitvariablen genommen werden, die reelle Werte enthalten. Diese *Uhren* laufen mit der gleichen Geschwindigkeit, d.h. sie divergieren nicht und können bei Transitionsübergängen auf 0 zurückgesetzt werden.

In Zustandsbeschriftungen werden Uhren verwendet, um Invarianten zu

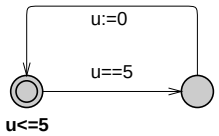


Abbildung 3.1: Automat mit Uhr

formulieren. Beispielsweise wird ein mit $u \leq 5$ beschrifteter Zustand *spätestens* verlassen, wenn die Uhr u den Wert 5 erreicht. Weiterhin werden Uhren in Transitionsbedingungen verwendet. Die Bedingung $u = 5$ lässt einen Transitionsübergang nur zum Zeitpunkt 5 zu, *erzwingt* ihn aber nicht. Nur die Kombination der Invariante $u \leq 5$ und der Transitionsbedingung $u = 5$ sichert ein Schalten genau zum Zeitpunkt 5 (Abb. 3.2). Danach wird die Uhr zurückgesetzt. Wie lange im rechten Zustand verweilt wird, ist unbestimmt. Potentiell wird dieser Zustand unendlich lange eingenommen.

Ein Zeitautomat enthält eine Menge reellwertiger Uhren C und eine Menge von Aktionen Σ. Eine Bedingung wird konjunktiv aus Beschränkungen von Uhren zusammengesetzt: $c \sim n$, mit $c \in C$, $\sim \in \{\leq, <, =, >, \geq\}$und $n \in \mathbb{N}$. $G(C)$ sei die Menge von Bedingungen. Im Folgenden werden knapp die zentralen Definitionen aus [AD96] zusammengefasst.

Definition Zeitautomat. Ein Zeitautomat ist ein Tupel $\langle N, l_0, E, I\rangle$ mit einer endlichen Menge N von Zuständen, einem Startzustand $l_0 \in N$, einer Menge von Transitionen $E \in N \times G(C) \times \Sigma \times 2^c \times N$ und einer Abbildung von Zuständen auf Invarianten: $I : N \rightarrow G(C)$. Transitionen werden mit $l \overset{g,a,r}{\rightarrow} l'$ bezeichnet, wenn $\langle l, g, a, r, l'\rangle \in E$. Dabei ist l der Quell- und l' der Zielzustand. Die Transition kann unter der Bedingung g genommen werden und löst dabei die Aktion a aus. Uhren aus der Menge r werden dabei auf Null zurückgesetzt.

Um komplexe Systeme zu modellieren, können sie in ihre *Prozesse* zerlegt dargestellt werden. Das Gesamtverhalten ergibt sich aus der parallelen Komposition, die sowohl das wechselseitige Auslösen von Aktionen in den Komponenten zulässt (Interleaving-Semantik), als auch die Synchronisation über Aktionen kennt.

Semantik von Zeitautomaten. Die Semantik eines Zeitautomaten wird über ein unendliches Transitionssystem definiert, in dem ein Zustand den aktuellen Kontrollzustand und die Werte der Uhren im System wiedergibt. Es gibt zwei Übergänge von einem Zustand des Transitionssystems in einen anderen [AD96]:

- *Ein zeitlicher Übergang.* Es vergeht Zeit, was sich darin äußert, dass die Werte aller Uhren um einen bestimmten Wert d synchron vergrößert werden. Es findet kein Wechsel des Kontrollzustandes statt. Sei l der aktuelle Zustand und u die gültige Belegung aller Uhren. Ein solcher Übergang kann nur stattfinden, wenn weiterhin die Zustandsinvariante gilt, d.h. sowohl u als auch $(u+d)$ erfüllen $I(l)$.[1] Formal wird dann ein solcher Übergang definiert als:

$$\langle l, u\rangle \overset{d}{\underset{}{\rightarrow}} \langle l, u+d\rangle$$

  mit $d \in \mathbb{R}_+$.

- *Eine Transition schaltet.* Die Werte der Uhren aus u bleiben unverändert, es sei denn, sie werden explizit zurückgesetzt, wenn sie in r enthalten sind: $u' = [r \mapsto 0]\, u$. Außerdem muss u die Transitionsbedingung g und u' die Invariante des Zielzustandes $I(l')$ erfüllen. Unter diesen Bedingungen löst die Transition $l \overset{g,a,r}{\underset{}{\rightarrow}} l'$ folgenden Übergang aus:

$$\langle l, u\rangle \overset{a}{\underset{}{\rightarrow}} \langle l', u'\rangle$$

Diese beiden Schritte können ausgehend von einem Startzustand beliebig oft ausgeführt werden, solange kein Deadlock auftritt.

$$\langle l_0, u_0\rangle \overset{d_1}{\rightarrow} \overset{a_1}{\rightarrow} \langle l_1, u_1\rangle \overset{d_2}{\rightarrow} \overset{a_2}{\rightarrow} \langle l_2, u_2\rangle \overset{d_3}{\rightarrow} \overset{a_3}{\rightarrow} \langle l_3, u_3\rangle \ldots$$

Besonders interessant bei der Analyse von Zeitautomaten ist die Erreichbarkeit von bestimmten Zuständen. Ein Zustand $\langle l, u\rangle$ ist erreichbar, wenn $\langle l_0, u_0\rangle \overset{*}{\underset{}{\rightarrow}} \langle l, u\rangle$. Verallgemeinert ausgedrückt ist ein Zustand l unter einer

[1] Aufgrund der Struktur der Zeitbedingungen gilt eine Invariante dann auch für alle zeitlichen Zwischenwerte.

Zeitbedingung ϕ erreichbar, wenn $\langle l, u \rangle$ erreichbar ist und u die Bedingung ϕ erfüllt. Mit Hilfe von Erreichbarkeitsprüfungen können auch Invarianten des Systems überprüft werden, indem die „verbotene“ Eigenschaft der Negation der Invarianten entspricht. Beispielsweise können so Sicherheitseigenschaften eines Systems analysiert werden.

3.1.2 Zeitautomaten in UPPAAL

Die theoretischen Erkenntnisse über Zeitautomaten wurden in funktionsfähige Implementierungen umgesetzt. Bedeutung haben vor allem die beiden Werkzeuge UPPAAL [LPY97] und Kronos [Yov97] erlangt. Insbesondere UPPAAL wurde seit der ersten Version im Jahre 1995 in einer Kooperation der Universitäten von Uppsala und Aalberg beständig weiterentwickelt.

Es ist mittlerweile mit einer komfortablen Oberfläche versehen, die Unterstützung bei der Modellierung, Simulation und Verifikation von Systemen bietet. Das Dateiformat wurde auf XML umgestellt, was den Datenaustausch vereinfacht. Vor allem sind die Verbesserungen der Performance zu Gute gekommen, so dass die Größe der verifizierbaren Modelle beständig gestiegen ist.

Ein Grund UPPAAL gegenüber anderen Werkzeugen den Vorzug zu geben, sind syntaktische Vorteile, welche die Modellierung vereinfachen. So können in Bedingungen nicht nur Uhren verwendet werden, sondern sie können auch mit Ausdrücken ganzzahliger Variablen gemischt werden. Weitere Besonderheiten sind *urgent Channels/Locations* und *committed Locations*. Dahinter verbergen sich Konstrukte, die einen sofortigen Transitionsübergang erzwingen. Sie werden weiter unten im Detail beschrieben. Außerdem bietet UPPAAL einiges an „syntaktischem Zucker“ wie z.B. Arrays von ganzzahligen Variablen und Uhren. Im Folgenden beschränken wir uns daher auf die Beschreibung von UPPAAL und konzentrieren uns auf die Konstrukte, die später verwendet werden.

UPPAAL modelliert Echtzeitsysteme als *Netzwerk von Zeitautomaten*. Jeder Zeitautomat repräsentiert einen *Prozess*, der parallel zu den anderen abläuft. Ein System von n parallelen Prozessen wird über einen Kompositionsoperator zu einem Gesamtsystem $A_1|\ldots|A_n$ verbunden. Dieser Operator entspricht dem CCS Operator für eine parallele Komposition. Die Kommunikation geschieht mit einer Synchronisation komplementärer Aktionen, die von außen nicht beobachtet werden kann. Es wird über einen Kanal a synchronisiert, wenn Transitionen mit den entgegengesetzten Aktionen $a!$ und

$a?$ gemeinsam schalten. Eine Aktion kann nur mit genau einer komplementären Aktion synchronisieren. Von außen ist diese Kommunikation nicht zu beobachten.

In neueren Versionen von UPPAAL ist diese Art der Synchronisation durch eine zweite ergänzt worden. In diesem Fall kann eine mit $a!$ beschriftete Transition mit beliebig vielen $a?$ synchronisieren, d.h. sie kann sowohl unabhängig von anderen Transitionen schalten, als auch zusammen mit unbeschränkt vielen $a?$. Daher wird diese Kommunikationsart bei UPPAAL als Broadcast bezeichnet. Umgekehrt benötigt ein $a?$ weiterhin ein zugeordnetes $a!$, um schalten zu können.

Die Kommunikationsarten werden in Abbildung 3.2 verdeutlicht. Seien die Prozesse A, B, C und D parallel geschaltet. Wenn a als einfacher Kanal ohne Broadcast benutzt wird, können folgende Prozesse zusammen schalten: $\{A, C\}$, $\{A, D\}$,$\{B, C\}$ oder $\{B, D\}$. Wird der Broadcast-Mechanismus verwendet, ergeben sich ausschließlich die folgenden Möglichkeiten der Synchronisation: $\{A, C, D\}$ oder $\{B, C, D\}$.

Prozess E kann allein nur schalten, wenn a als Broadcast-Kanal gekennzeichnet ist. Allerdings reicht eine erfüllte Bedingung nicht aus, um zu erzwingen, dass die Transition tatsächlich genommen wird. Es ist aber in UPPAAL möglich, einen Kanal als *urgent* zu kennzeichnen. In diesem Fall schaltet die Transition ohne Zeitverzug, sobald die Bedingung `true` wird. An solchen Transitionen sind allerdings keine Bedingungen zulässig, die Uhren enthalten, da sich dieses nicht mit dem Berechnungsmodell von UPPAAL verträgt. Kanäle, die urgent sind, sind besonders hilfreich, wenn sie mit Bedingungen über ganzzahlige Variablen verknüpft sind. Nur so lässt sich eine sofortige Reaktion auf die Änderung von ganzzahligen Variablen realisieren.

UPPAAL erlaubt es, Felder von verschiedenen Konstrukten wie Variablen, Uhren und Konstanten zu definieren. Es können auch Felder von Kanälen definiert werden. Mit $b[1]$ wird dann ein Kanal b mit dem Index 1 bezeichnet. Die Prozesse F und G schalten in Abbildung 3.2 nur gemeinsam, wenn $i = j$ gilt. Andernfalls schaltet nur F, wenn b ein Broadcast-Kanal ist. Ist b ein einfacher Kanal, kann keine Transition schalten.

Da der Index auch durch eine Variable angegeben werden kann, ergeben sich Möglichkeiten zur prägnanteren Modellierung. Beispielsweise kann der Index mit einer Kennnummer eines bestimmten Prozesses übereinstimmen, an den man so Anfragen adressieren kann.

Neben der synchronen Kommunikation über Kanäle gibt es in UPPAAL die Möglichkeit, asynchron über *globale Variablen* zu kommunizieren. Ande-

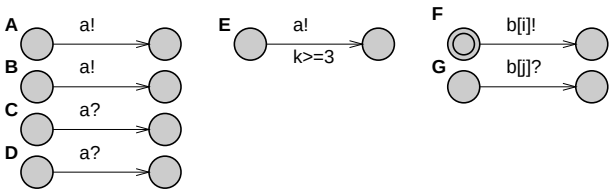


Abbildung 3.2: Kommunikation über Kanäle

rerseits können Variablen auch lokal zu einem bestimmten Prozess definiert werden.

In [Möl02] wird die Semantikdefinition von oben an die Version der Zeitautomaten von UPPAAL angepasst. Dabei steht der Aspekt der Aufteilung von Zeitautomaten in kommunizierende Prozesse im Vordergrund. Im Folgenden werden die Definitionen dieser Quelle genutzt, um die UPPAAL-Semantik zu beschreiben.

Die Semantik eines Netzwerks von Zeitautomaten wird definiert, indem ein Transitionssystem definiert wird, dessen Zustände *Vektoren* von Kontrollzuständen enthalten: $\langle \bar{l}, u \rangle$. Es gibt wiederum die Möglichkeit, entweder einen Übergang zu einem anderen Zustand des Transitionssystems zu machen, indem man Zeit verstreichen lässt (synchrones Vergrößern der Uhrenwerte) oder es kann eine Transition des Zeitautomaten schalten, wodurch Aktionen ausgelöst werden.

Die Regel für das Verstreichen von Zeit ist sehr ähnlich der eines Zeitautomaten ohne parallele Prozesse. Im Unterschied zu oben muss allerdings für jede Komponente i überprüft werden, ob die neuen Zeitwerte der Uhren die Invarianten erfüllen, d.h. $(u + d) \in \bigwedge I(l_i)$. Unter dieser Bedingung ist dann folgender analoger Übergang möglich [Möl02]:

$$\langle \bar{l}, u \rangle \xrightarrow{d} \langle \bar{l}, u + d \rangle$$

Da in UPPAAL eine Transition ohne Synchronisation oder zwischen komplementären Aktionen $a!$ und $a?$ nicht von außen zu beobachten ist, wird die unsichtbare Aktion τ eingeführt. Existiert eine lokale Transition $l_i \xrightarrow{g,\tau,r} l_i'$, die *keine* zu synchronisierende Aktion enthält, führt ein Übergang zu einer lokalen Aktualisierung des Kontrollvektors ($\bar{l}' = \bar{l}[l_i/l_i']$) und zum Zurücksetzen der Uhren in r ($u' = [r \mapsto 0]u$). Diese Schritte finden unter der Vorausset-

zung statt, dass die Transitionsbedingung erfüllt ist ($u \in g$) und nach einem Übergang keine Invariante verletzt ist ($u' \in I(l'_i) \wedge \bigwedge_{k \neq i} I(l_k)$). Unter diesen Bedingungen ergibt sich der Übergang:

$$\langle \bar{l}, u \rangle \xrightarrow{\tau} \langle \overline{l'}, u' \rangle$$

Findet eine Synchronisation zwischen zwei Prozessen statt, ergibt sich ein Übergang gleicher Form. Voraussetzung sind zwei Transitionen $l_i \xrightarrow{g_i, a!, r_i} l'_i$ und $l_j \xrightarrow{g_j, a?, r_j} l'_j$, die von aktiven Zuständen in unterschiedlichen Prozessen ausgehen und über den Kanal a synchronisieren können. Im Kontrollvektor werden beide Zustände durch ihre jeweiligen Nachfolger ausgetauscht ($\overline{l'} = \bar{l}[l_i/l'_i, l_j/l'_j]$) und die Uhren zurückgesetzt ($u' = [r_i \cup r_j \mapsto 0]u$), wenn die Transitionsbedingungen erfüllt sind ($u \in g_i \wedge g_j$) und die neuen Invarianten Gültigkeit haben ($u' \in I(l'_i) \wedge I(l'_j) \bigwedge_{k \neq i,j} I(l_k)$).

Neben Uhren gibt es in UPPAAL auch ganzzahlige Variablen. Beide Sorten von Variablen können in Feldern zusammengefasst werden. Ganzzahligen Variablen und Felder können an Transitionen neue Werte in Form von Ausdrücken zugewiesen werden. Um diese Erweiterung in der Semantik zu erfassen, reicht es, die Zuweisungsfunktion u auf die benutzten ganzzahligen Variablen auszuweiten. Da ganzzahlige Variablen nicht nur auf 0 aktualisiert werden, muss bei synchronisierten Transitionen eine Reihenfolge festgelegt werden. In UPPAAL ist festgelegt, dass die Aktualisierungen auf der Seite von $a!$ Vorrang vor denen auf der Seite von $a?$ haben. Damit ergibt sich zunächst $u* = ([r_i \mapsto f(u)]u)$ und dann $u' = ([r_j \mapsto f(u*)]u*)$. Besteht ein r_x aus mehreren Zuweisungen, werden diese von links nach rechts ausgewertet, wobei jeweils auf die dann gültige Variablenbelegung Bezug genommen wird und Änderungen sofort wirksam werden.

Ebenso wie Kanäle können Zustände in UPPAAL als *Urgent* oder als *Committed* gekennzeichnet werden. Urgent-Zustände werden ohne Zeitverlust wieder verlassen. Eine auf 0 initialisierte Zeitvariable c und die Invariante $c <= 0$ leisten dasselbe.

Committed-Zustände hingegen lassen sich mit den vorhandenen Konstrukten nicht so einfach nachbilden. Sie erzwingen nicht nur das zeitlose Verlassen eines Zustands, sondern geben darüber hinaus von solchen Zuständen abzweigenden Transitionen Priorität über andere Transitionen. Sie schal-

ten sozusagen etwas„schneller als zeitlos“. Alle Committed-Zustände werden verlassen, bevor andere Zustände mit deren Transitionen betrachtet werden.

Sei $C(\bar{l})$ die Menge der Committed-Zustände im Kontrollvektor. Im Transitionssystem kann nur dann ein zeitlicher Übergang erfolgen, wenn $C(\bar{l}) = \emptyset$. Analoges gilt auch für Urgent-Zustände.

Eine Transition im Zeitautomaten eines Prozesses kann nur erfolgen, wenn $C(\bar{l}) = \emptyset$ oder wenn der Zustand, der verlassen wird, committed ist, d.h. $l_i \in C(\bar{l})$. Wenn sich zwei Prozesse über einen Kanal synchronisieren, reicht es, wenn einer der beiden Quellzustände committed ist. Das heißt, eine synchrone Aktion kann erfolgen, wenn $C(\bar{l}) = \emptyset \vee l_i \in C(\bar{l}) \vee l_j \in C(\bar{l})$.

Verifikationsanfragen. Für die Definition von Verifikationsanfragen verwendet UPPAAL, wie viele Werkzeuge, temporallogische Ausdrücke. Es wird eine Untermenge von TCTL unterstützt, in der Quantoren nur auf der obersten Ebene zu finden sind. Es werden die Pfadquantoren A (entspricht *für alle Pfade*) und E (*für mindestens einen Pfad*) und innerhalb von Pfaden die Quantoren $[\,]$ (*in jedem Zustand, immer*) und $\langle\,\rangle$ (*in mindestens einem Zustand, irgendwann*) benutzt. Seien P und Q boolesche Ausdrücke, die sich auf Zustände, Variablen und Uhren beziehen. Dann sind folgende Ausdrücke verifizierbar:

- $A[\,]\ P$: Es muss auf allen Pfaden immer die Eigenschaft P gelten (Abb. 3.3 oben links). Ist diese Eigenschaft verletzt, liefert UPPAAL einen Pfad zu $\neg P$.

- $E\langle\,\rangle\ P$: Es gibt einen Pfad, so dass irgendwann P gilt (Abb. 3.3 oben rechts). UPPAAL kann einen Beispielpfad zu P generieren.

- $A\langle\,\rangle\ P$: Es muss auf allen Pfaden irgendwann P gelten (Abb. 3.3 unten links). Diese Eigenschaft wird widerlegt, sobald eine Schleife gefunden wird, in der P nicht auftritt.

- $E[\,]\ P$: Es gibt einen Pfad, auf dem immer die Eigenschaft P gilt (Abb. 3.3 unten Mitte). Diese Eigenschaft wird bewiesen, sobald ein Pfad, der immer P aufweist, in einer Schleife endet.

- $P \rightarrow Q$: Es handelt sich um eine Kurzform für $\forall\Box(P \Rightarrow \forall\Diamond Q)$, d.h. nach einem P tritt zwangsläufig auf jedem Pfad ein Q auf (Abb. 3.3 unten rechts).

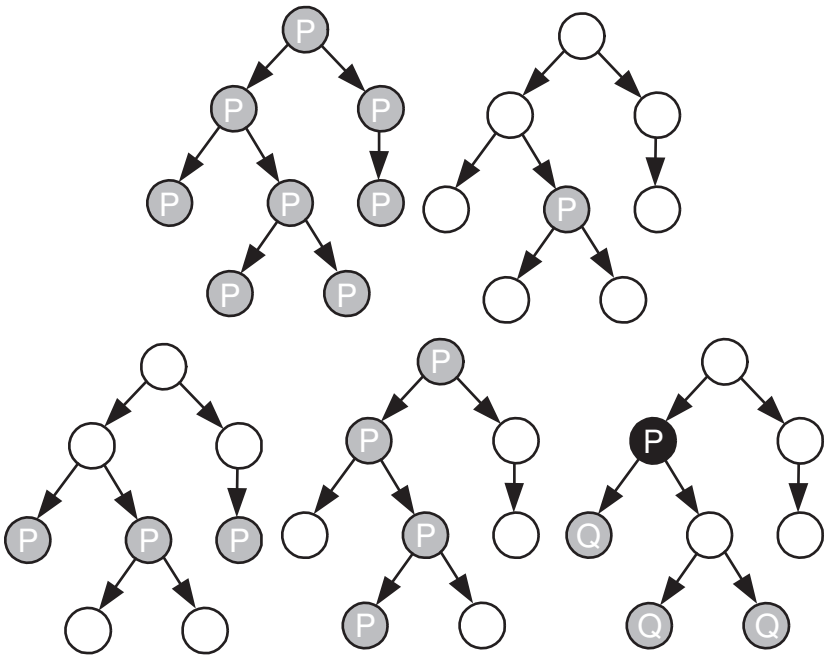


Abbildung 3.3: Temporallogische Ausdrücke

Die Eigenschaften $A[\,]\ P$ und $E\langle\,\rangle\ P$ sind eng verwandt:

$$A[\,]\ P \Leftrightarrow \neg E\langle\,\rangle\ \neg P$$

Sie werden als *Safety-Eigenschaften* bezeichnet, da sie benutzt werden können, um zu spezifizieren, dass ein System keine „verbotenen“ bzw. sicherheitskritische Zustände einnimmt. Weiterhin können *bounded Liveness-Eigenschaften* überprüft werden. Darunter werden Eigenschaften verstanden, die fordern, dass etwas innerhalb einer gewissen Zeit passiert sein muss. Dazu kann eine boolesche Hilfsvariable eingeführt werden, mit der vermerkt wird, ob das geforderte Ereignis schon eingetreten ist. Zur Überprüfung von bounded Liveness wird dann die Invariante verifiziert, dass entweder dieses Ereignis schon eingetreten ist oder die zeitliche Grenze noch nicht erreicht wurde.

Die anderen drei Eigenschaften $A\langle\,\rangle\ P$, $E[\,]\ P$ und $P \rightarrow Q$ gehören zur Klasse der *unbounded Liveness-Eigenschaften*. Sie erfordern bei der Verifikation generell einen höheren Aufwand.

Es konnte hier nur ein knapper Überblick über Zeitautomaten gegeben werden. Für eine ausführliche Beschreibung sei auf [AD94, AD96] verwiesen. Die semantischen Besonderheiten von UPPAAL werden in [Möl02] behandelt. UPPAAL selbst wird in [LPY97] vorgestellt. Allerdings ist diese Beschreibung mittlerweile überholt. Die aktuellen syntaktischen Möglichkeiten entnimmt man oft am besten der Online-Hilfe des Werkzeugs.

3.2 Abbildung von Systembeschreibungen

Wir beschäftigen uns zunächst mit der Transformation von Systemen, die mit UML-Zustandsdiagrammen modelliert werden, in Zeitautomaten. Bei einer Verifikation führt die Einführung jeder neuen Variable (zeitlich-kontinuierlich oder ganzzahlig) zu einer Vergrößerung des Zustandsraums. Daher sollte eine Implementierung mit der Einführung neuer Variablen sehr zurückhaltend sein. Die im Folgenden vorgestellte Abbildung von UML-Konstrukten in Zeitautomaten geht auf Optimierungsaspekte bezüglich der Reduzierung des Zustandsraum nur am Rande ein. Beispielsweise wird konzeptionell für jede Zeitmarke jeweils eine neue Uhr eingeführt, auch wenn die Zeitmarke nur an einer Stelle verwendet wird. In der Implementierung wurden Optimierungsmöglichkeiten stärker berücksichtigt, auch wenn dieses an dieser Stelle nicht

thematisiert wird, da es vor allem um die Darstellung des Konzeptes der Abbildung in Zeitautomaten geht. Dieses gilt auch für den nächsten Abschnitt 3.3.

3.2.1 Beschreibung der Übersetzung

Die Übersetzung erfolgt in drei Phasen: Zunächst wird Hierarchie aus den Zustandsdiagrammen entfernt. Dies ist erforderlich, da Zeitautomaten keine hierarchischen Konstrukte unterstützen. Danach werden die flachen Zustandsautomaten in ein System von Zeitautomaten übersetzt. Im letzten Schritt werden Automaten ergänzt, die den Ablauf zwischen den generierten Automaten im Sinne der Semantik koordinieren.

3.2.1.1 Beseitigung von Hierarchie

Für jedes Zustandsdiagramm $sc_{obj} =< S_{basic} \cup S_{composite} \cup S_{final}, I, T, L, child >$ bestehend aus der Zustandsmenge S, den Startzuständen I, den Transitionen T, den Transitionsbeschriftungen L und der Funktion *child* für Unterzustände wird ein äquivalentes flaches Zustandsdiagramm $sc'_{obj} =< S', I', T', L', child' >$ erzeugt. Dabei bleiben die einfachen Zustände und die Endzustände erhalten ($S' = S_{basic} \cup S_{final}$) und kein Zustand verfügt über Unterzustände: $\forall s \in S' : child'(s) = \emptyset$. Folgende Schritte werden zum Entfernen der Hierarchie durchgeführt:

Startzustand. Die Funktion $init^*$ bildet einen zusammengesetzten Zustand auf einen eindeutigen Startzustand ab, indem sie rekursiv vom Wurzelknoten an die Startzustände ermittelt, bis dieses kein zusammengesetzter Zustand mehr ist. Ein flaches Zustandsdiagramm enthält nur den Startzustand $I' = \{init^*(root)\}$.

Behandlung von Transitionsbeschriftungen. Bevor wir die zusammengesetzten Zustände aus den Zustandsdiagrammen entfernen, müssen wir explizite Uhren für zeitgetriggerte *after*-Transitionen einfügen. *After*-Transitionen beziehen sich auf den Zeitpunkt des Betretens eines Zustands. Wenn eine solche Transition von einem zusammengesetzten Zustand ausgeht, ist nach dessen Entfernen unklar, zu welchem Zeitpunkt die Zeitmessung beginnt.

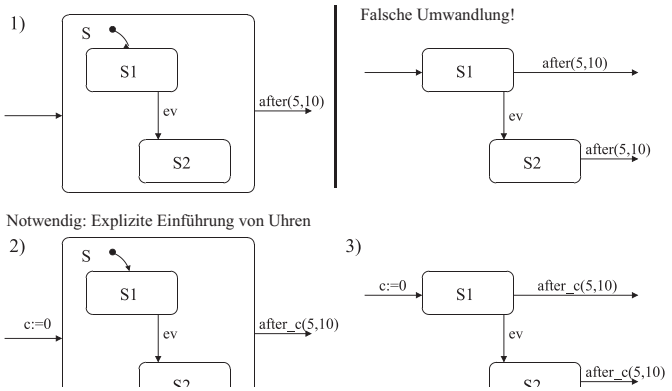


Abbildung 3.4: Einführung expliziter Uhren für *after*-Transitionen

Daher werden alle zusammengesetzten Zustände ermittelt, die von einer *after*-Transition verlassen werden. An jeder Transition, die als Ziel einen solchen zusammengesetzten Zustand oder einen seiner Unterzustände hat, wird eine Uhr c explizit zurückgesetzt. An den *after*-Transitionen wird eine Referenz auf diese Uhr eingefügt, indem Trigger der Form $after(min, max)$ durch $after_c(min, max)$ ersetzt werden (Abb. 3.4). Damit bleibt der Bezug zwischen dem Zurücksetzen der Uhren und den *after*-Transitionen auch nach Entfernung der Oberzustände erhalten.

Transitionen. Es werden alle Transitionen übernommen, mit Ausnahme der Transitionen, die einen zusammengesetzten Zustand verlassen oder betreten.

- Eine Transition, die einen zusammengesetzten Zustand verlässt und einen expliziten Trigger aufweist, wird für jeden untergeordneten, einfachen Zustand kopiert. Dieser Zustand wird ihr als Ausgangszustand zugewiesen.

- Wenn eine Transition zu einem zusammengesetzten Zustand führt, kann diese Transition auf denjenigen einfachen Zustand umgeleitet werden, den man erhält, wenn man rekursiv die untergeordneten Default-Transitionen verfolgt.

- Wenn eine *triggerlose* Transition einen zusammengesetzten Zustand

verlässt, wird dessen Endzustand zu ihrem Quellzustand gemacht. Alle anderen Transitionen werden nicht verändert.

Die UML-Semantik gibt Transitionen Priorität, deren Quellzustände sich weiter unten in der Zustandshierarchie befinden. Um dieses Prioritätsschema beizubehalten, müssen alle Transitionen, die von zusammengesetzten Zuständen abzweigen, und die sich im Konflikt (gleiches auslösendes Ereignis) mit Transitionen einer höheren Priorität befinden, mit der Konjunktion der Negation der Bedingungen dieser Transitionen versehen werden.

Damit sind alle Transitionen ersetzt worden, die von zusammengesetzten Zuständen ausgehen. Daher können zusammengesetzte Zustände nun entfernt werden.

3.2.1.2 Übersetzung von flachen Zustandsdiagrammen nach UPPAAL

Wir gehen jetzt davon aus, dass ein flaches Zustandsdiagramm vorliegt, aus dem alle hierarchischen Zwischenzustände bereits entfernt wurden. Es soll nun das flache Zustandsdiagramm $sc_{obj} =< S, I, T, L, child >$ in ein äquivalentes System von Zeitautomaten $ta_{obj} =< S', I', T', L', V', C' >$ (mit den ganzzahligen Variablen V und der Uhrenmenge C) übersetzt werden. Dieses wird für jedes Objekt $obj \in Obj$ durchgeführt. Dabei wird die Struktur der Zustandsdiagramme so weit wie möglich erhalten, d.h. alle Zustände, der Initialzustand und die Transitionen bleiben erhalten, wenn dies möglich ist. Wir bezeichnen das Ergebnis dieser Transformation im Folgenden als *Objektautomat*.

Ganzzahlige Variablen und Uhren. Zusätzlich zu der Variablenmenge V, die bereits in den Zustandsdiagrammen verwendet und übernommen wird, werden für jedes Zustandsdiagramm drei neue Variablen eingeführt:

- Die Variable in_{obj} enthält ein gesendetes Ereignis für ein Zustandsdiagramm, bevor es in dessen Warteschlange eingereiht wird. Ein Ereignis wird dabei durch einen konstanten Wert repräsentiert.
- Die Variable out_{obj} speichert ein gerade aktuelles Ereignis, während es verarbeitet wird.
- Die boolesche Variable $compl_{obj}$ signalisiert das Auftreten eines Completion-Ereignisses.

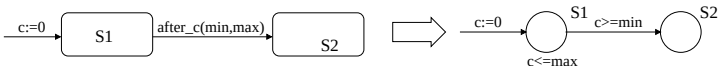


Abbildung 3.5: Übersetzung von *after*-Transitionen

Damit ergibt sich die Variablenmenge

$$V' = V \cup \{in_{obj}, out_{obj}, compl_{obj} | obj \in Obj\}.$$

Die Menge der Uhren C wird ebenfalls aus dem Zustandsdiagramm übernommen und später um eine Uhr für die globale Zeit und um weitere Uhren zur Umsetzung von *after*-Transitionen erweitert.

***After*-Transitionen.** Transitionen, die durch *after*-Ereignisse getriggert werden, werden, wie in Abb. 3.5 gezeigt, übersetzt. Mit Hilfe einer neuen Uhr c, die vor allen eingehenden Transitionen zurückgesetzt wird, einer Invariante $c \leq max$ am Zustand s_1 und einer Bedingung $c \geq min$ an der zeitgetriggerten Transition kann das Verhalten nachgebildet werden[2]. Wenn eine *after*-Transition über eine weitere Bedingung g verfügt, kann es sein, dass die von s_1 ausgehende Transition nicht schalten kann. Aufgrund der Invariante würde es zu einem Deadlock kommen, wenn die Maximalzeit der *after*-Transition überschritten wird. Dieser Deadlock entspricht nicht dem Verhalten auf UML-Ebene. Daher wird eine Kopie des Zustandes s_1 ohne Invariante und ohne Abbild der *after*-Transition erstellt. In diesem Zustand ist gleiches Verhalten wie im Zustand s_1 möglich. Lediglich der Zeitpunkt des Schaltens der *after*-Transition wurde verpasst. Eine Transition mit der Bedingung $\neg g \wedge (c \geq min)$ von s_1 zum neuen Zustand stellt sicher, dass es zu keinem Deadlock kommt.

Ereignisse. Zeitlich getriggerte Ereignisse wurden bereits behandelt. Alle anderen Transitionen werden entweder von expliziten oder Completion-Ereignissen $\surd$ ausgelöst. Auf Ebene der Zeitautomaten wird das Schalten von Transitionen durch die Einführung von Bedingungen, die sich auf die

[2]Es wird an dieser Stelle darauf verzichtet, ein zeitliches Ereignis in eine Warteschlange einzureihen. Wegen der zugrunde gelegten Semantik ist eine Warteschlange immer leer, wenn ein zeitliches Ereignis ausgelöst wird und diese Ereignisse können daher immer direkt verarbeitet werden.

Ereignisse beziehen, beschränkt. Das aktuell aktive Ereignis ist dabei als spezifische Konstante jeweils in der Variable out_{obj} gespeichert. Vorhandene Bedingungen werden übernommen:

$$guard'(t) = \begin{cases} guard(t) \wedge out_{obj} == ce & if\ event(t) = \surd \\ guard(t) \wedge out_{obj} == ev & if\ event(t) = ev \neq \surd \end{cases}$$

Wenn als Transitionseffekt das Versenden eines Ereignisses vorgesehen wird, wird auf Ebene der Zeitautomaten die Aktionsliste um das Setzen der Eingabevariable des Zielobjektes ergänzt.

$$actions(t) \cup \{in_{obj} := ev\}$$

Completion-Ereignisse. Ein Completion-Ereignis ist einem Zustand zugeordnet und kann nur von diesem Zustand ausgehende Transitionen auslösen. Ist in diesem Moment die Transitionsbedingung nicht erfüllt, wird es nicht an Transitionen mit niedrigerer Priorität weitergereicht, wie es bei anderen Ereignissen üblich ist. Trotzdem wird im Folgenden nicht zwischen verschiedenen Completion-Ereignissen unterschieden. Der Grund dafür ist, dass triggerlose Transitionen, die von zusammengesetzten Zuständen ausgehen, nach unserer Konstruktion in flachen Zustandsdiagrammen von den Endzuständen abzweigen, die ihrem zusammengesetzten Zustand zugeordnet sind. Damit bleibt die Zuordnung zwischen Completion-Ereignis und zugehörigem zusammengesetzten Zustand implizit erhalten, ohne dass es einer Differenzierung bedarf. Wird ein Completion-Ereignis ausgelöst, wird daher die objektspezifische boolesche Variable $compl_{obj}$ auf *true* gesetzt. Dieses passiert beim Betreten eines Endzustandes oder eines einfachen Zustandes.

Kontrolliertes Schalten der Transitionen. Jede Transition des ursprünglichen Zustandsdiagramms muss mit einem Kontrollautomaten synchronisiert werden (vgl. Abschnitt 3.2.1.3). Dieser Kontrollautomat sorgt für ein Zusammenspiel der Komponenten, das der Semantik gerecht wird. Daher wird an jeder zeitgetriggerten Transition eine objektbezogene Synchronisationsaktion $sy_{obj}!$ eingetragen. An den durch explizite oder Completion-Ereignisse ausgelösten Transitionen wird die umgekehrte Kommunikationsrichtung $sy_{obj}?$ verwendet. Über die Aktion $sy_{obj}?$ kann der Kontrollautomat das Verarbeiten

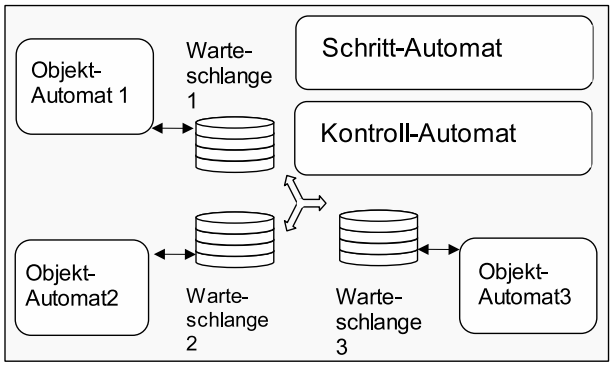


Abbildung 3.6: Übersicht Abbildung in Zeitautomaten

eines Ereignisses bewirken. Über die Aktion $sy_{obj}!$ hingegen kann der Kontrollautomat über ein zeitliches Ereignis informiert werden. Da es zur Übersetzungszeit aufgrund der Transitionsbedingungen nicht festzustellen ist, ob eine Transition tatsächlich schaltet, muss diese Synchronisationsaktion vom UPPAAL-Typ *Broadcast* sein[3]. Ansonsten würde der Kontrollautomat blockiert werden und ein Deadlock auftreten, wenn er über sy_{obj} synchronisiert und keine Transition im Objektautomat schalten kann.

3.2.1.3 Übersetzung eines Systems von Zustandsdiagrammen

Bisher wurden einzelne Zustandsdiagramme übersetzt. Es müssen noch weitere Komponenten ergänzt werden, die das Zusammenspiel der Objektautomaten gemäß der Semantik sicherstellen. In Abbildung 3.6 wird eine Übersicht über die beteiligten Automaten gegeben. Wir beschreiben im Folgenden

- einen Automaten, der eine Ereigniswarteschlange für eine asynchrone Kommunikation realisiert,
- einen Kontrollautomaten, der die Zusammenarbeit zwischen den Objektautomaten koordiniert, indem er solange Ereignisse an die Objektautomaten zur Behandlung weiterreicht, bis alle Warteschlangen leer sind,
- und einen Schrittautomaten, der nach jedem Schritt notwendige Datenoperationen durchführt.

[3]Nicht blockierende Synchronisationsaktion zwischen $a!$ und $a?$. Die Aktion $a!$ kann beliebig viele Empfänger $a?$ triggern (vgl. 3.1).

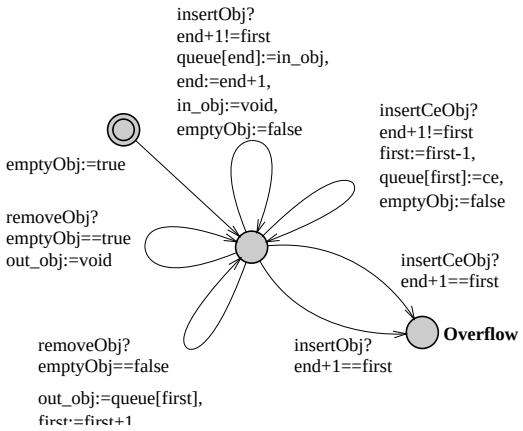


Abbildung 3.7: Warteschlange für Ereignisse

Ereigniswarteschlangen. Obwohl UPPAAL nur synchrone Kommunikation unterstützt, lässt sich asynchrone Kommunikation durch Einführung von Automaten, die Warteschlangen verwalten, nachbilden. Für jedes Objekt $obj \in Obj$ wird ein solcher Automat erzeugt. Dieser Automat realisiert folgende Operationen

- $insert(in_{obj}, obj)$ fügt den Wert der Eingangsvariable in_{obj} an das Ende der entsprechenden Warteschlange.
- $insertCe(obj)$ fügt ein Completion-Ereignis an den *Anfang* der Warteschlange.
- $remove(obj)$ entfernt das erste Ereignis aus der Warteschlange und überträgt den repräsentierenden Wert in die Ausgangsvariable out_{obj}

Abbildung 3.7 zeigt eine mögliche Implementierung in Form eines Zeitautomaten. UPPAAL erlaubt als Datenstruktur Felder von ganzzahligen Werten. Damit können nur Warteschlangen mit einer begrenzten Länge realisiert werden. Ein Einfügen eines neuen Ereignisses in eine volle Warteschlange führt in den Zustand $Overflow$. Ereignisse werden in das erste freie Feld der Warteschlange eingereiht. Das Löschen bzw. Einfügen eines Completion-Ereignisses bewirkt ein Verschieben aller anderen Ereignisse der Warteschlange. Das Einfügen eines Ereignisses, das in in_{obj} zwischengespeichert wird,

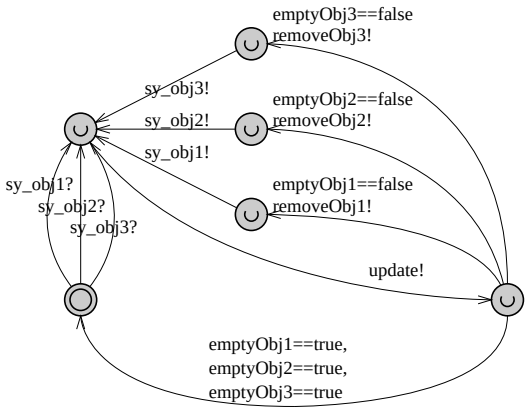


Abbildung 3.8: Kontrollautomat

wird über die Synchronisationsaktion *insertObj* ausgelöst. Ein Completion-Ereignis wird mit *insertCeObj* an die erste Position gesetzt. Die boolesche Variable *emptyObj* gibt an, ob eine Warteschlange noch Ereignisse enthält. Eine Synchronisation über *removeObj* bewirkt ein Entfernen des ersten Elements der Warteschlange. Dieses Ereignis steht dann in der Variable für das gegenwärtig gültige Ereignis out_{obj} zur Verfügung.

Zeitliche Ereignisse werden nicht über die Warteschlange verwaltet. Wenn ein zeitgetriggertes Ereignis auftritt, befindet sich ein System von Zustandsdiagrammen immer in einem stabilen Zustand mit leeren Warteschlangen, da in der instabilen Phase alle Ereignisse konsumiert werden. In diesem Fall kann ein Ereignis direkt eine Transition triggern und braucht nicht über eine Warteschlange zwischengespeichert werden.

Kontrollautomat. Neben den Objektautomaten wird noch ein weiterer Automat benötigt, der den Ablauf eines Schrittes koordiniert (Abb. 3.8). Dieser Automat kommuniziert mit den Objektautomaten $obj \in Obj$ über die Kanäle sy_{obj}. Der Startzustand bezeichnet den stabilen Zustand. Dieser Zustand wird verlassen, wenn in einem Objekt eine zeitgetriggerte Transition schaltet. Dazu wird über den Kanal sy_{obj} synchronisiert. Der Kontrollautomat aktualisiert die Eingabe- und Ausgabevariablen der Ereigniswarteschlangen, indem er über den Kanal *update* einen weiteren Automaten aktiviert (siehe Schrittautomat).

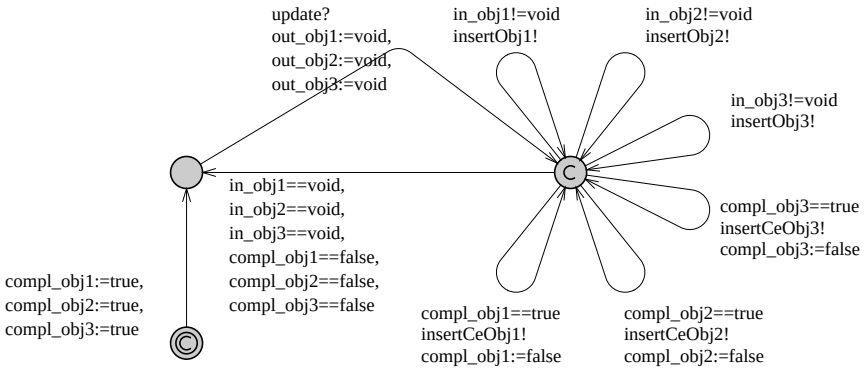


Abbildung 3.9: Automat für Aktualisierung nach einem Schritt

Solange es noch Ereignisse in den Warteschlangen gibt, wird eines der betroffenen Objekte nichtdeterministisch ausgewählt und das erste Ereignis der Warteschlange in die Variable out_{obj} verschoben. Über die synchrone Aktion sy_{obj} - mit umgekehrter Synchronisationsrichtung - wird dann der zugehörige Objektautomat zum Schalten verlasst. Dort können als Reaktion neue Ereignisse verschickt werden. Diese Änderungen werden nach jedem Schritt durch den Schrittautomaten umgesetzt. Sobald alle Warteschlangen leer sind, wechselt der Kontrollautomat in den Wartezustand.

Schrittautomat. Um die Konstruktion übersichtlicher zu halten, werden die Datenoperationen nach einem Schritt in einem gesonderten Automaten realisiert (Abb. 3.9). Der Automat verhält sich wie eine Methode, die über den Kanal *update* aufgerufen wird. Es werden dann die folgenden Operationen ausgeführt:

- Die Variablen out_{obj} mit den gegenwärtig aktiven Ereignissen jedes Objekts werden zurückgesetzt.
- Die Variableninhalte von in_{obj} werden in die entsprechenden Warteschlangen kopiert. Die Variablen werden danach ebenfalls zurückgesetzt.
- Completion-Ereignisse werden an den *Anfang* einer Warteschlange kopiert, wenn $compl_{obj} = true$ ist.

- Wenn alle Werte in die Warteschlangen übernommen wurden, kehrt der Automat in den Wartezustand zurück.

Jedes Objekt kann nur ein eintreffendes Ereignis pro Objekt speichern. Daher kann an einer Transition einem bestimmten Objekt maximal ein Ereignis geschickt werden. Es ist möglich, mehreren Objekten jeweils ein Ereignis zu schicken. Mit einer Serie von Transitionen können unbegrenzt viele Nachrichten verschickt werden. Daher spielt diese Einschränkung in der Praxis keine Rolle und wird zugunsten einer effizienteren Darstellung in Zeitautomaten in Kauf genommen.

3.2.2 Beispiel

Die oben beschriebene Transformation wird am Beispiel einer Ampelsteuerung verdeutlicht. Dabei werden eine Fußgängerampel $AmpelFg$ und eine Ampel für Fahrzeuge $AmpelFz$ (Abb. 3.10 unten) von einer Steuerung (Abb. 3.10 oben) zyklisch angesteuert. Die Ampeln werden von der Steuerung eingeschaltet. Danach wird ein Wechsel der Ampelphasen teils von der Steuerung getriggert, teils wird ein Wechsel auch durch einen lokalen Timeout verursacht. Da die Steuerung (nicht modellierte) Umwelteinflüsse berücksichtigt, gibt es Varianzen im zeitlichen Ablauf.

Teile der automatischen Umsetzung dieses Systems sind in den Abbildungen 3.11 bis 3.13 dargestellt. Es fehlen der Kontrollautomat, der Schrittautomat und drei Automaten zur Verwaltung der Ereigniswarteschlangen. Diese Automaten sind wie im vorherigen Abschnitt beschrieben aufgebaut.

In Abbildung 3.11 lässt sich erkennen, wie Unterzustände den Namen ihrer übergeordneten Zustände als Präfix erhalten. Zustände ohne Zeitbedingungen werden vom Kontrollautomat getriggert. Zustände mit Zeitbedingungen aktivieren dagegen den Kontrollautomat. Dies ist an der Umkehrung der Synchronisationsrichtung der mit $sy\_AmpelFz$ beschrifteten Aktionen erkennbar. Da der Zustand An über einen Endzustand verfügt, kann die triggerlose Transition zum Zustand Aus nur genommen werden, wenn dieser Endzustand betreten wird.

Der Zustand $Warten$ der Steuerung wird über eine zeitlich getriggerte Transition verlassen. Daher wird diese Transition mit allen Unterzuständen

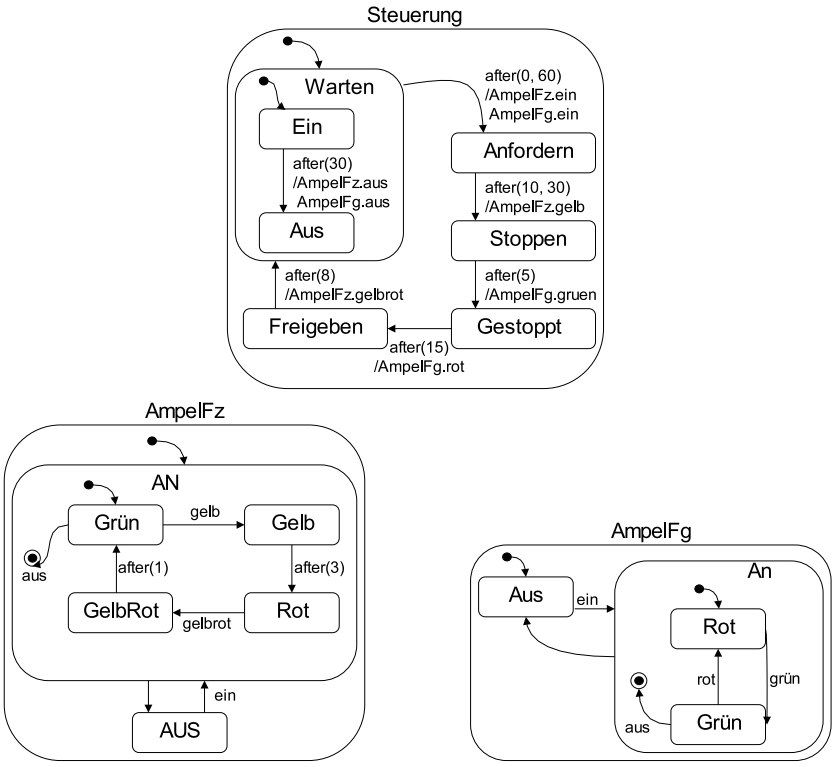


Abbildung 3.10: Ampelsteuerung, Fahrzeug- und Fußgängerampel

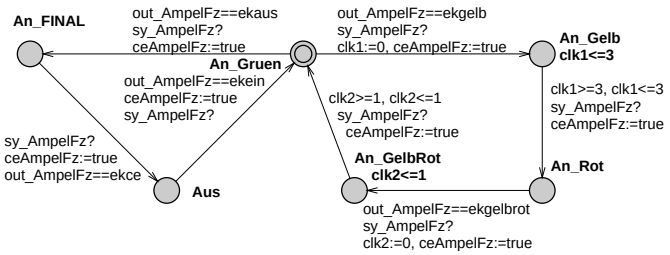


Abbildung 3.11: Automat für die Fahrzeugampel

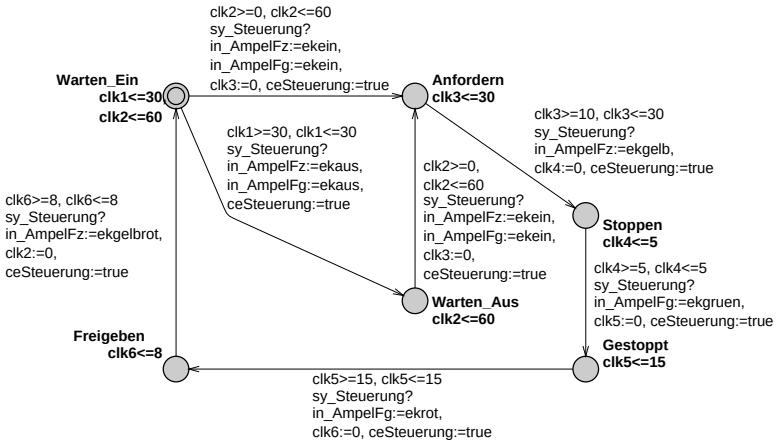


Abbildung 3.12: Automat für die Ampelsteuerung

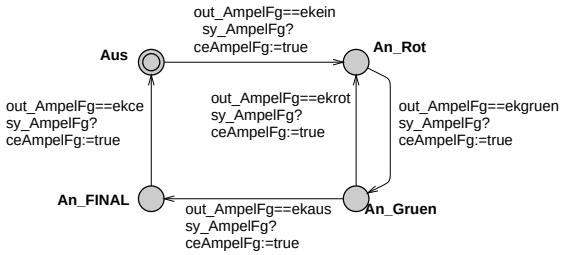


Abbildung 3.13: Automat für Fußgängerampel

verbunden (Abb. 3.12). Die Steuerung versendet zahlreiche Ereignisse, die zunächst in den Variablen $in\_AmpelFg\_$ und $in\_AmpelFz\_$ zwischengespeichert werden. Von dort werden sie in die entsprechenden Ereigniswarteschlangen eingefügt. Nach der Entnahme durch den Schrittautomaten stehen sie für einen Schritt in den Variablen $out\_AmpelFg$ und $out\_AmpelFz$ zur Verfügung und können in Bedingungen benutzt werden.

3.3 Abbildung der Anforderungsbeschreibung

In diesem Abschnitt wird beschrieben, wie Anforderungen, die als Sequenzdiagramme formuliert sind, in eine analysierbare Form überführt werden. Dabei ist nicht nur eine Transformation der Sequenzdiagramme notwendig.

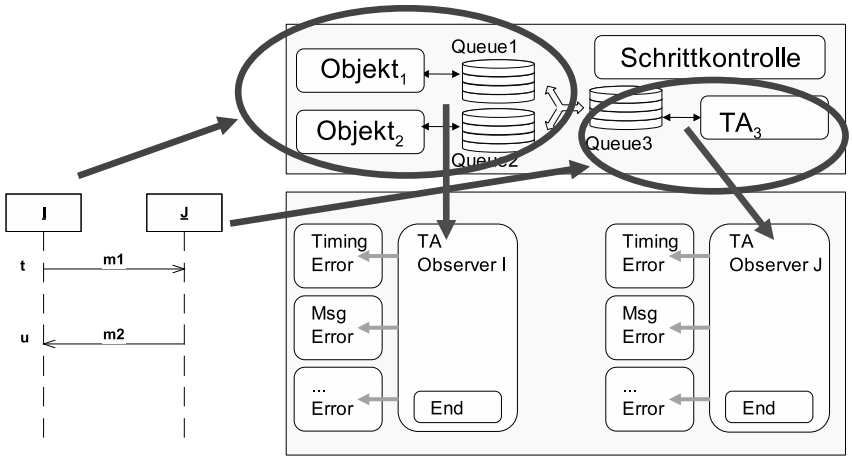


Abbildung 3.14: Zuordnung von Anforderung und System

Auch das Systemmodell muss für die Analyse aufbereitet werden. Das Systemmodell wurde bereits in die Notation der Zeitautomaten überführt (siehe Abschnitt 3.2). Für eine Anforderungsanalyse müssen relevante Ereignisse auch auf der Analyseebene der Zeitautomaten beobachtbar gemacht werden. Zusammengefasst sind folgende Schritte durchzuführen (Abb. 3.14):

- Das System wird observierbar gemacht. Dabei darf weder der Systemablauf, noch das zeitliche Verhalten verändert werden.
- Aus Sequenzdiagrammen werden Observerautomaten erzeugt. Diese Automaten repräsentieren, inwieweit gefordertes Verhalten zu einem bestimmten Zeitpunkt während der Laufzeit erfüllt wurde. Abweichendes Verhalten verzweigt zu Fehlerzuständen.
- Die zugehörigen Anforderungen werden in temporaler Logik erzeugt. Beispielsweise wird die Erreichbarkeit von Fehlerzuständen überprüft.

Analyseparameter. Ein besonderer Vorteil des hier verfolgten Ansatzes ist die Flexibilität der Analyse. Bezüglich der Sequenzdiagramme kann unter zwei zeitlichen Ordnungen und zwei Kommunikationsmechanismen gewählt werden. Es ist damit auch möglich, Systeme, die direkt in Zeitautomaten

spezifiziert wurden, als Systemmodell aufzufassen. Es werden sowohl ein synchrones wie ein asynchrones Kommunikationsmodell unterstützt. Zusammen mit dem Anforderungsstatus eines bestimmtes Sequenzdiagramms ergeben sich damit jeweils unterschiedliche Ereignisabfolgen, die von einem System von Observern beobachtet werden. Das heißt, die *Ordnung der Ereignisse*, die *Kommunikationsform* und der *Status eines Sequenzdiagramms* sind die Parameter, welche die Konstruktion des Observersystems beeinflussen.

Die Modifikation der Systembeschreibung zur Observierung hängt insbesondere von der *Zuordnung* der in den Sequenzdiagrammen benutzten Instanzen und Nachrichten zu den entsprechenden Realisierungen ab. Diese ist von der gewählten Darstellung des Systems (Zustandsdiagramme, Zeitautomaten) abhängig. Darüber hinaus bestimmt die gewählte *Kommunikationsform* die Realisierung der Ereignisse. Diese zwei Parameter bestimmen die notwendigen Anpassungen des Systems.

Nach der Konstruktion des analysierbaren Modells müssen Verifikationsanfragen formuliert werden. Die temporalen Anforderungen in Form der Logik, wie sie UPPAAL verwendet, lassen sich aus den Observerautomaten und dem *Anforderungsstatus der Sequenzdiagramme* ableiten. In den folgenden Abschnitten wird auf die Parameter der Analyse näher eingegangen.

3.3.1 Zuordnung und Anforderungsstatus

Es werden zwei unterschiedliche Möglichkeiten zugelassen, ein System zu spezifizieren, welches den Ausgangspunkt für eine automatische Überprüfung von Anforderungen darstellt:

- Die direkte Spezifikation mit Zeitautomaten und synchroner Kommunikation oder
- eine Menge von Zustandsdiagrammen in UML-Notation.

Die Transformation von Zustandsdiagrammen in Zeitautomaten wurde bereits im Abschnitt 3.2 beschrieben. In diesem Fall ergibt sich durch die Einführung von Ereigniswarteschlangen für Signale eine asynchrone Kommunikation. Als Vorbereitung für eine Konsistenzprüfung von Anforderungen und Systemmodell müssen diese beiden Sichten verknüpft werden. Es wird eine Abbildung benötigt, die Instanzen von Sequenzdiagrammen ihren Realisierungen im System zuordnet. Weiterhin ist eine Abbildung erforderlich,

die einen Zusammenhang zwischen Nachrichten in Sequenzdiagrammen und Kommunikationsmechanismen in der Systembeschreibung herstellt. Dabei ist prinzipiell jeder Kommunikationsmechanismus nutzbar, der sich auf der Ebene von Zeitautomaten ausdrücken lässt. Wichtig für die Umsetzung ist die Unterscheidung zwischen asynchroner und synchroner Kommunikation, da dieses zu Unterschieden bei der Observerkonstruktion führt. In dieser Arbeit werden beispielhaft zwei Kommunikationsmechanismen besprochen. Dabei handelt es sich um synchrone Kommunikation über Kanäle innerhalb von Zeitautomaten und asynchrone Kommunikation über Ereigniswarteschlangen, wie sie sich aus der UML-Spezifikation ergeben. Die Zuordnungsfunktionen werden in den folgenden Definition zusammengefasst:

Definition *Zuordnungsfunktionen.*

- $Part(I)$ ist die Menge der Zeitautomaten oder Zustandsdiagramme, welche die Instanz I repräsentieren.
- $Com(m)$ ist die Realisierung der Nachricht m, d.h. ein Kanal (synchron) oder ein UML-Signal (asynchron).

Es wird ebenfalls eine eindeutige Umkehrung $Part^{-1}$ benötigt: Da eine Instanz später zur Konstruktion eines Observers führt, wird die Umkehrung der Zuordnungsfunktionen benötigt, um einen relevanten Observer vom Auftreten eines Ereignisses zu benachrichtigen. Dabei sind nur Zuordnungsfunktionen $Part$ zugelassen, die injektiv sind. Fälle, in denen ein Zustandsdiagramm oder Zeitautomat mehrere Instanzen repräsentiert, werden ausgeschlossen. Analog dazu wird Com^{-1} benötigt, um ein Kommunikationsereignis im System einer Nachricht in einem Sequenzdiagramm zuzuordnen. Wie darüber hinaus noch der Sender und Empfänger einer Nachricht bestimmt wird, wird im Detail weiter unten beschrieben. Dabei muss bei einem Systemmodell bestehend aus entweder Zeitautomaten oder Zustandsdiagrammen jeweils unterschiedlich vorgegangen werden, da

- bei Zeitautomaten die Kommunikation über synchrone Aktionen die Ermittlung des Empfängers zur Spezifikationszeit ausschließt, wohingegen Ereignisse in Zustandsdiagrammen gerichtet sind und
- bei Zustandsdiagrammen normalerweise der Sender eines Signals beim Empfang unbekannt ist, wohingegen bei synchroner Kommunikation sich Sender und Empfänger leicht ermitteln lassen.

Daher gestaltet sich die Umsetzung der Zuordnung von System und Anforderungen dieser beiden Kommunikationsarten unterschiedlich. Weitere Unterschiede treten bei der Observerkonstruktion auf (vgl. Abschnitt 3.3.3). Die Observer repräsentieren in ihren Zuständen den Fortschritt in einem bestimmten Szenario. Jeder Zustand kann einer registrierten Abfolge von Ereignissen zugeordnet werden. Ist ein beobachtetes Verhalten zulässig, dann wird ein Zielzustand erreicht, andernfalls ein Fehlerzustand. Weiterhin müssen alle zeitlichen Bedingungen und Einschränkungen bzgl. der Schleifeniterationen eingehalten werden. Damit ergeben sich die grundlegenden Anforderungen bezüglich der Observer:

- Der Zielzustand muss immer erreichbar sein.
- Es darf nie ein Fehlerzustand betreten werden.

Bei optionalem Anforderungsstatus entfällt der letzte Punkt, weil lediglich gefordert ist, dass *ein* Lauf des Systems das Sequenzdiagramm erfüllt. Die automatische Erzeugung dieser Forderungen wird später im Detail beschrieben.

3.3.2 Zeitliche Ordnung

Die Konstruktion eines Observers verläuft für Sequenzdiagramme mit unterschiedlichem Status weitgehend ähnlich. Sequenzdiagramme, deren Status *mandatory* ist, beginnen unverzüglich mit der Beobachtung des Systems, wohingegen Sequenzdiagramme mit dem Status *optional* zu beliebigen Startzeiten eine erfüllende Ereignisfolge erwarten. An dieser Stelle wird die grundsätzliche Konstruktion von Observern erläutert, die in allen Fälle gleich ist.

Ein System vollführt während seines Ablaufs eine Abfolge von Kommunikationsereignissen, die durch einen Namen, der zugeordneten Instanz und eine Kommunikationsrichtung gekennzeichnet sind:

Definition *Systemereignis*. Das Ereignis $m_{P\rightarrow}$ bezeichne ein Ereignis mit dem Namen m (ergibt sich aus dem Namen des Kommunikationsträgers, beispielsweise eines Kanals oder eines Signals), das im Prozess P der Systembeschreibung stattfindet. Der Pfeil $\rightarrow$ kennzeichnet es als Empfangsereignis. Entsprechend handelt es sich bei $m_{P'\leftarrow}$ um ein Empfangsereignis. Gehören $m_{P\rightarrow}$ und $m_{P'\leftarrow}$ zu einer Nachricht m, werden sie als *Komplementärereignisse* bezeichnet.

In einem synchronen Modell finden komplementäre Empfangs- und Sendeereignisse zeitgleich statt. In diesem Fall lassen sich komplementäre Ereignisse einfach zuordnen und einem Observer können Sender, Empfänger und Name der Kommunikation gemeinsam übermittelt werden. Im asynchronen Fall von Zustandsdiagrammen werden hier Annahmen über die Realisierung des Kommunikationssystems gemacht, um diese eindeutige Zuordnung zu gewährleisten. Es wird davon ausgegangen, dass Signale zuverlässig unter Beibehaltung ihrer Reihenfolge übertragen werden. Spontan auftretende Signale werden ebenso ausgeschlossen, wie Signale, die verloren gehen. Da es sich bei dem betrachteten Kommunikationsmechanismus um die Warteschlangen aus Abschnitt 3.2 handelt, sind diese Annahmen als korrekt anzusehen.

Die Observierung von Ereignissen erfolgt zum Teil instanzweise. Dieses motiviert die folgende Definition:

Definition *Einschränkung auf eine Instanz.* Sei a eine Ereignisfolge. Dann bezeichnet a/I eine *Einschränkung auf eine Instanz* I, wenn alle Ereignisse e mit $Inst(e) \neq I$ durch ε ersetzt werden. In ähnlicher Weise wird eine Menge E von Ereignissen auf I eingeschränkt: $E/I := \{e \in E | Inst(e) = I\}$.

Die Menge der möglichen Ereignisfolgen, die sich aus einem Diagramm ergibt, ist abhängig von der Ordnung, die bei der Interpretation eines Sequenzdiagramms zugrunde gelegt wird. Die wohl bekannteste Ordnung ist die *visuelle Ordnung*, die für MSCs [Ren98] angenommen wird:

- Ein Sendeereignis einer Nachricht findet vor dem komplementären Empfangsereignis statt.
- Alle Ereignisse einer Instanz I sind total geordnet. Je später ein Ereignis stattfindet, desto weiter unten befindet es sich auf der Lebenslinie.

Da Ereignisse, die verschiedenen Instanzen zugeordnet sind, nicht geordnet sein müssen, handelt es sich nur um eine partielle Ordnung.

Definition *Visuelle Ordnung.* Sei M eine Folge von Nachrichten. Für jede Instanz $I \in Inst(M)$ sei die Relation $<_{M,I} \subseteq Ev(M)/I \times Ev(M)/I$ die implizierte Ordnung der Ereignisse bezogen auf die Instanz I. Dann ist

$$<_M := \bigcup_{I \in Inst(M)} <_{M,I} \cup \{(s,e) \in Ev(M) \times Ev(M) | (s,e)\ komplementär\}$$

die durch M implizierte *visuelle Ordnung*. Dabei sind komplementäre Ereignisse das Sende- und Empfangsereignis einer Nachricht.

Abgesehen von dieser Ordnung sind beliebige weitere Ordnungen denkbar. In dieser Arbeit wurde eine weitere Ordnung untersucht, welche die visuelle Ordnung verschärft. Dabei wird gefordert, dass zwischen dem Sende- und Empfangsereignis einer Nachricht keine Ereignisse anderer Nachrichten liegen. Da sich Nachrichten nicht mehr überschneiden, ergibt sich eine partielle Ordnung auf Nachrichten. Anschaulich beschrieben geht diese Ordnung davon aus, dass in einem System jeweils nur eine Nachricht unterwegs ist, die vollständig abgearbeitet werden muss, bevor die folgende Nachricht versendet wird.

Nachrichten, die sich auf disjunkte Instanzen beziehen, sind zunächst nicht geordnet. Es wird daher weiterhin gefordert, dass alle Nachrichten *innerhalb* eines Diagramms total geordnet sind, auch wenn sie sich auf unterschiedliche Instanzen beziehen. Die Ordnung auf Nachrichten ist vorgegeben.[4] Damit sind in einem Sequenzdiagramm natürlich auch alle Ereignisse total geordnet, auch wenn sie sich auf unterschiedlichen Instanzen befinden. Diese Ordnung wird beispielsweise in [FHD+99] zugrunde gelegt, da dort eine visuelle Ordnung angenommen wird, aber nicht geordnete Nachrichten ausgeschlossen werden.

Die Motivation für die verschärfte Ordnung ist, dass sich der Zustandsraum bei einer Verifikation drastisch verkleinert, wenn sie anwendbar ist. Ein Beispiel für ein reales Anwendungsgebiet ist die aktuelle Implementierung des asynchronen Modus des *Industrial Automation Protocols (IAP)* [Bec01]: Soll neben den isochronen Sensordaten ein außergewöhnliches Kontrollereignis (z. B. Ausfall eines Sensors) übertragen werden, wird auf den asynchronen Modus zurückgegriffen. Bisher ist das IAP auf *eine* asynchrone Nachricht pro Zyklus eingeschränkt, d. h. Komplementärereignisse von asynchronen Nachrichten finden immer paarweise statt.

Zusammenfassend beschrieben stehen folgende Ordnungen zur Auswahl:

- Es gilt die *visuelle Ordnung*. Alle verbliebenen Ereignispaare sind ungeordnet.
- Es wird eine totale Ordnung auf Nachrichten eines Diagramms angenommen. Wenn außerdem vorausgesetzt wird, dass sich Nachrichten

[4] In dieser Arbeit werden Sequenzdiagramme als Abfolge von Nachrichten zwischen Instanzen abgespeichert. Damit wird eine Ordnung auf Nachrichten impliziert. Wird nur die visuelle Ordnung angenommen, wird die Nachrichtenordnung ignoriert.

nicht überlappen, ergibt sich eine totale Ordnung auf Kommunikationsereignisse. Diese Ordnung wird als *erweiterte Ordnung* bezeichnet.

3.3.2.1 Ereignisfolgen bei erweiterter Ordnung

Wenn wir eine erweiterte Ordnung annehmen, sind alle Ereignisse total geordnet[5]. Die Menge der möglichen Ereignisfolgen $Ef(M)$ ergibt sich aus der totalen Anordnung M der überlappungsfreien Nachrichten. Sende- und Empfangsereignis einer Nachricht erfolgen immer paarweise. Alternative Verzweigungen führen zu unterschiedlichen Ereignisfolgen. Sei M ein Ausdruck für die erlaubten Abfolgen von Nachrichten. Dieser Ausdruck kann aus der Modellierung in Form von Sequenzdiagrammen abgeleitet werden. Dann ergeben sich folgende Ereignisfolgen:

$$Ef(M) := \begin{cases} SendEv(M), EmphEv(M) & falls\ M\ Nachricht \\ ef\left(M_1\right), ef\left(M_2\right) & falls\ M = M_1, M_2 \\ ef\left(M_1\right) | ef\left(M_2\right) & falls\ M = M_1 | M_2 \\ \varepsilon & falls\ M = \varepsilon \end{cases}$$

Die Eigenschaften dieser Ordnung ermöglichen eine vereinfachte Konstruktion eines Observers. Aufgrund der totalen Ordnung innerhalb einer Alternative lässt sich ein globaler Observer für alle Instanzen erzeugen. Dieses ist im komplexeren Fall der visuellen Ordnung so nicht möglich.

3.3.2.2 Ereignisfolgen bei visueller Ordnung

Da die visuelle Ordnung nur eine partielle Ordnung ist, lässt sie eine große Anzahl von tatsächlichen Ausführungen zu, die sich aus den Permutationen ungeordneter Ereignisse ergeben. Ein globaler Observer würde entsprechend unübersichtlich werden. Daher wird davon abgesehen, einen solchen globalen Observer explizit zu konstruieren. Statt dessen wird für die Observerkonstruktion ausgenutzt, dass Ereignisse entlang der Lebenslinien von Sequenzdiagrammen immer total geordnet sind. Dabei bleibt die zeitliche Anordnung der Komplementärereignisse zunächst unberücksichtigt.

[5]Dieses gilt nicht für Ereignisse, die sich in unterschiedlichen alternativen Verzweigungen befinden. Alternativen können z. B. durch bedingte Sequenzdiagramme oder Schleifen auftreten. In diesem Fall ist eine Ordnung nicht vorhanden, aber auch nicht sinnvoll, da derartige Ereignisse nicht gemeinsam in einem Ablauf auftreten.

Es stellt sich die Frage, ob die Observer verknüpft werden müssen, um sicherzustellen, dass ein Empfangsereignis einer Nachricht immer nach deren Versenden auftritt. Dies hängt davon ab, ob im beobachteten System spontane Empfangsereignisse auftreten und ob Nachrichten verloren gehen. An dieser Stelle wird die generierte Systembeschreibung aus Abschnitt 3.2 zugrunde gelegt. Dort werden Ereignisse beobachtet, die nicht spontan auftreten und zuverlässig weitergeleitet werden. Daher wäre im Observersystem eine derartige Prüfung redundant.

Im allgemeinen Fall müsste man ein Sendeereignis speichern, um es als Vorbedingung für einen Empfang einer Nachricht überprüfen zu können, indem beispielsweise eine entsprechende boolesche Variable gesetzt wird. Da der Zustandsraum der Verifikation empfindlich auf die Einführung neuer Variablen reagiert, wird hier ausgenutzt, dass dank der Eigenschaften der generierten Kommunikationsmechanismen (Ereigniswarteschlangen)[6] darauf verzichtet werden kann.

Wenn M eine Abfolge von Nachrichten ist, lässt sich die Ordnung der Ereignisse, die einer Instanz zugeordnet sind, direkt aus der Spezifikation ablesen. Sie entspricht der oben beschriebenen lokalen Ordnung $<_{M,I}$.

Die zwei Ordnungen, die zur Interpretation von Sequenzdiagrammen zugelassen sind, führen also zu unterschiedlichen Konstruktionen zur Observierung:

- Die erweiterte Ordnung erlaubt durch die totale Ordnung von Nachrichten und Ereignissen die Konstruktion eines globalen Observers.
- Die visuelle Ordnung führt zu einem System von Observern. Jeder Observer ist einer Instanz zugeordnet und beobachtet deren lokale Ereignisse.

3.3.2.3 Zeitbedingungen

Zeitbedingungen verknüpfen Zeitmarkierungen. Eine Zeitmarkierung ist einem Ereignis zugeordnet und bezeichnet den Zeitpunkt von dessen Auftreten. Eine Zeitbedingung enthält ein bis zwei Zeitmarkierungen.

[6] Dieses gilt nur insoweit, wenn kein Überlauf der Ereigniswarteschlange stattfindet. Jede endliche Warteschlange kann überlaufen, wenn das Versenden von Signalen in einem ungünstigen Verhältnis zu deren Verarbeitung steht. Dieses wird aber in jedem Fall als Designfehler betrachtet. Ein entsprechend eingefügter Fehlerzustand wird auf Erreichbarkeit geprüft.

- Wird *eine* Zeitmarkierung verwendet, lassen sich Bedingungen an den absoluten Zeitpunkt des Auftretens des zugeordneten Ereignisses formulieren.
- Im gängigeren Fall von *zwei* Zeitmarken lassen sich zeitliche Differenzen zwischen zwei Ereignissen spezifizieren.

Zur Überprüfung von Zeitbedingungen werden auf Ebene der Zeitautomaten Uhren eingeführt. Es ist wichtig, wann im Ablauf diese Zeitbedingungen überprüft werden. Dieses muss direkt nach Auftreten eines relevanten Ereignisses geschehen, ohne dass zuvor Zeit vergeht. Die Realisierung auf der Ebene von Zeitautomaten werden wir im nächsten Abschnitt betrachten. Zunächst muss jedoch das Ereignis bestimmt werden, dem wir die Zeitbedingungen zur Überprüfung zuordnen.

Bei Zeitbedingungen mit nur einer Zeitmarke wird die Zeitbedingung diesem Ereignis zugeordnet. Andernfalls muss sie dem zur *Laufzeit* später auftretenden Ereignis zugeordnet werden. Wird die visuelle Ordnung angenommen, kann das spätere Ereignis im allgemeinen Fall nicht zur Übersetzungszeit ermittelt werden.

Definition. Eine Zeitbedingung T heißt *dynamisch*, wenn T zwei Zeitmarken enthält, die visuelle Ordnung gilt und die betroffenen Ereignisse nicht geordnet sind. Andernfalls heißt T *statisch*. Die Zuordnung einer Zeitbedingung zu einem Ereignis ergibt sich wie folgt:

Sei f eine Ereignismenge und T eine Zeitbedingung. Sei $<$ eine zeitliche Ordnung auf f. Zu jeder Zeitmarke t aus T sei e_t das zugeordnete Ereignis. Dann bezeichnet $StatEv_f(T)$ das Ereignis, dem T zur statischen Prüfung zugeordnet werden kann bzw. $DynEv_f(T)$ zwei Ereignisse, an denen eine dynamische Überprüfung von T stattfinden muss. $Var\,(T)$ bezeichne dabei die Menge der Zeitmarken in T.

$$StatEv_f\,(T) := \begin{cases} \{e_t\} & falls\, Var\,(T) = \{t\}\,, e_t \in Ev\,(f) \\ \{e_u\} & falls\, Var\,(T) = \{t,u\}\,,\ e_t < e_u,\ e_u \in Ev\,(f) \\ \emptyset & sonst \end{cases}$$

$$DynEv_f\,(T) := \begin{cases} \{e_t, e_u\} \cap Ev\,(f) & falls\, Var\,(T) = \{t,u\}\,,\ e_t,\ e_u\, ungeordnet \\ \emptyset & sonst \end{cases}$$

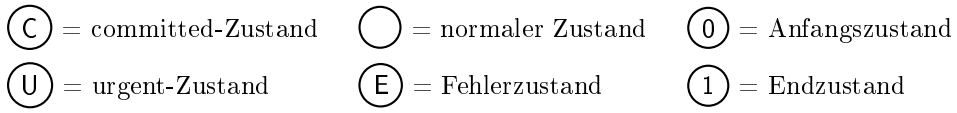


Abbildung 3.15: Zustände von Zeitautomaten

Gemäß dieser Definition können die Ereignisse bestimmt werden, denen eine Zeitbedingung zugeordnet werden muss und es wird bestimmt, ob es sich um eine statische oder dynamische Prüfung handelt.

3.3.3 Synchrone und asynchrone Kommunikation

Nachdem in den vorherigen Abschnitten vorbereitende Definitionen gemacht wurden, indem die von einer zeitlichen Ordnung abhängigen Ereignisfolgen und die Zuordnung der Prüfung von Zeitbedingungen zu Ereignissen eingeführt wurden, wird nun die eigentliche Abbildung in Zeitautomaten beschrieben. Dazu werden Details der Observierung und die Konstruktion der Observerautomaten spezifiziert. Wir haben es dabei mit drei Phasen zu tun:

- Zunächst muss ein Ereignis e in einem System erkannt werden. Dieses Ereignis trete in der Form $Sys(e)$ als Transitionsbeschriftung auf. Andere Bestandteile der Beschriftung spielen keine Rolle und werden unverändert übernommen.
- Um das System observierbar zu machen, müssen Kommunikationsaktionen mit dem Observer ergänzt werden. Zu diesem Zwecke muss $Sys(e)$ durch $Obs(e)$ ersetzt werden.
- Ein Observer muss ein Ereignis registrieren können. Er überprüft, ob ein zulässiges Ereignis innerhalb vorgegebener Zeitbedingungen empfangen wurde. Bei einer Verletzung einer Anforderung wird in einen entsprechenden Fehlerzustand verzweigt.

Um die unterschiedlichen Zustände von Zeitautomaten zu kennzeichnen, verwenden wir die Notation aus Abbildung 3.15.

Der allgemeine Aufbau eines Observers ist in Abbildung 3.16 dargestellt. Für jedes Ereignis, das erwartet wird, wird ein Zustand eingefügt, in dem

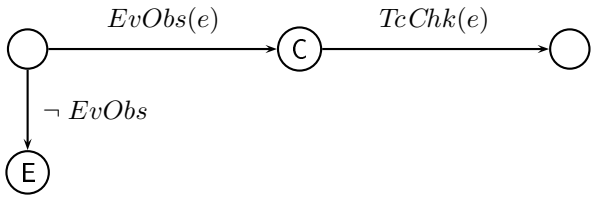


Abbildung 3.16: Observer für ein Ereignis

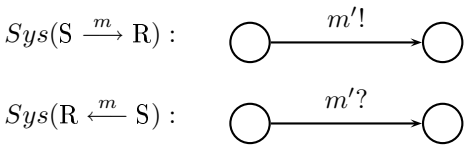


Abbildung 3.17: Synchrone Kommunikation

auch Zeit vergehen kann. Die mit $EvObs(e)$ bezeichnete Transition überprüft das Auftreten des erwarteten Ereignisses. Andernfalls wird in einen Fehlerzustand verzweigt. Danach werden ggf. zeitliche Bedingungen, die dem Ereignis e zugeordnet sein können, überprüft. Damit diese Überprüfung durchgeführt werden kann, werden an Ereignissen, die in Zeitbedingungen enthalten sind, entweder Zeitvariablen zurückgesetzt oder die Bedingung direkt überprüft. Details werden weiter unten in einem Abschnitt zur Überprüfung von Zeitbedingungen ausgearbeitet.

Synchrone Kommunikation. Synchrone Kommunikation findet in unserem Kontext über Kanäle von Zeitautomaten statt. Abbildung 3.17 zeigt eine Sendeaktion (oben) und eine Empfangsaktion (unten). Es wird die Abbildung

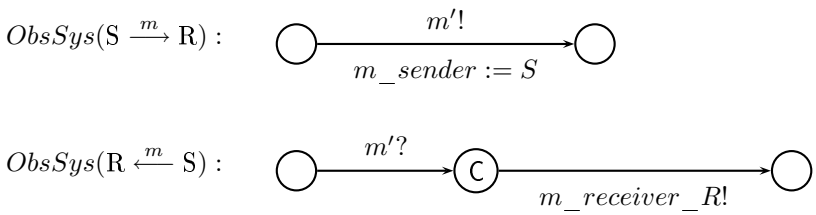


Abbildung 3.18: Observierte synchrone Kommunikation

Part genutzt, um von einer Instanz zu deren implementierenden Automaten zu gelangen. Gehören diese Ereignisse im System zu einer Nachricht m, die von S nach R geschickt wird, ist die obere Transition in einem Automaten $S' \in Part(S)$ und die untere Transition in einem Automaten $R' \in Part(R)$ enthalten. Das heißt, R, S und m sind Elemente in den Sequenzdiagrammen und R', S' und m' deren Gegenstücke in den Zeitautomaten.

Für den Kommunikationsträger gilt $m' = Com(m)$. In diesem Fall sind bei der Synchronisation der Sender S, der Empfänger R und die Nachricht m bekannt.

Eine Observierung erfolgt über eine synchrone Aktion mit dem Observer (Abb. 3.18). Diese Aktion ergibt sich per Namenskonvention aus der Nachricht m und der empfangenden Instanz R. Diese Aktion wird immer unmittelbar nach dem Empfang eines Ereignisses ausgelöst. Der Sender S wird dem Observer über eine globale Variable m_sender mitgeteilt.

Der Observer ist als Gegenstück zur Observierung aus Abbildung 3.18 aufgebaut. Dort werden ggf. auch Uhren zurückgesetzt, falls der Auftrittszeitpunkt des Ereignisses Teil einer Zeitbedingung ist.

Asynchrone Kommunikation. Die Umsetzung von asynchroner Kommunikation auf Ebene von Zeitautomaten wurde in Abschnitt 3.2 beschrieben. An dieser Stelle ist das Auftreten von Ereignissen relevant, d. h. das Versenden und Empfangen von Signalen. Die Umsetzung von Sende- und Empfangsereignissen wird in Abbildung 3.19 verdeutlicht. Gesendet an bzw. Empfangen von einer Instanz I wird über die Variablen I_{in} und I_{out}. Diese Variablen bilden die Schnittstelle zur Ereigniswarteschlange, deren Verwaltung im letzten Abschnitt besprochen wurde. Abbildung 3.20 zeigt den Zugriff auf diese Variablen: Oben in der Abbildung setzt die Instanz S mit $R'_{in} := m'$ die Eingabevariable der Instanz R auf m', d. h. sie versendet die Nachricht m.[7] Unten wird ein Empfangsereignis durch Verwendung der Variable R'_{out} in einer Transitionsbedingung verarbeitet.

Die Observierung dieser Ereignisse erfolgt ähnlich wie im synchronen Fall über Kanäle, die den Kommunikationsträger m (d. h. den Nachrichtennamen), die Art des Ereignisses (*sender* oder *receiver*) und die ausführende

[7] Hier gilt wie im synchronen Fall: S, R und m bezeichnen Elemente in Sequenzdiagrammen; S', R' und m' benennen die zugehörigen Elemente in den Zeitautomaten.

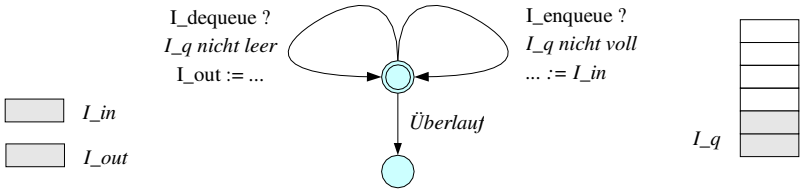


Abbildung 3.19: Kommunikationsschema für asynchrone Kommunikation

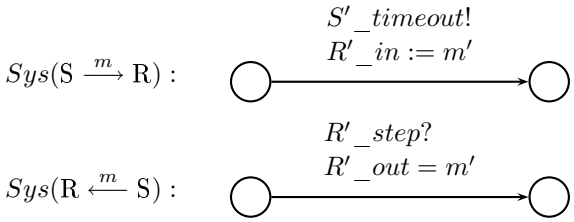


Abbildung 3.20: Ereignisse in Zeitautomaten

Instanz im Namen beinhalten (Abb. 3.21). Beim Senden wird das Zielobjekt dem Observer über die Variable $m\_receiver$ mitgeteilt. Anders als im synchronen Fall liegen Sende- und Empfangsereignis zeitlich auseinander. Dem Empfänger eines asynchronen Signals ist im allgemeinen Fall der Sender unbekannt. Soll auch der Sender observiert werden, muss seine Bezeichnung der Nachricht hinzugefügt werden. In diesem Fall muss die Warteschlange aus Abschnitt 3.2 so umgebaut werden, dass sie zwei Einträge pro Nachricht zulässt: Den Nachrichtennamen und zusätzlich den Absender.[8] Für den Sender wird dann eine zweite Eingabevariable (hier R'_{inS}) eingeführt, über die der Sendername der Warteschlange übergeben wird. Über R'_{outS} kann der Sender beim Empfänger abgerufen und dem Observer übergehen werden.[9]

Werden an einer Transition mehrere Ereignisse versendet, müssen auch mehrere Observer benachrichtigt werden. Dazu findet die Kommunikation mit den Observern sequentiell statt (siehe Abb. 3.22). Da pro Transition ma-

[8] Diese Erweiterung ist leicht durchzuführen, zieht allerdings einen erheblich größeren Zustandsraum bei der Verifikation nach sich. Kann auf die Verifikation des Senders beim Empfang verzichtet werden, lässt sich in der Implementierung die Verdoppelung der Warteschlange per Parameter ausstellen.

[9] Alternativ dazu kann der Sender in der Ereignisbenennung kodiert werden. Aufgrund des erweiterten Wertebereichs für Ereigniseinträge ist der Effekt auf den Zustandsraum vergleichbar negativ.

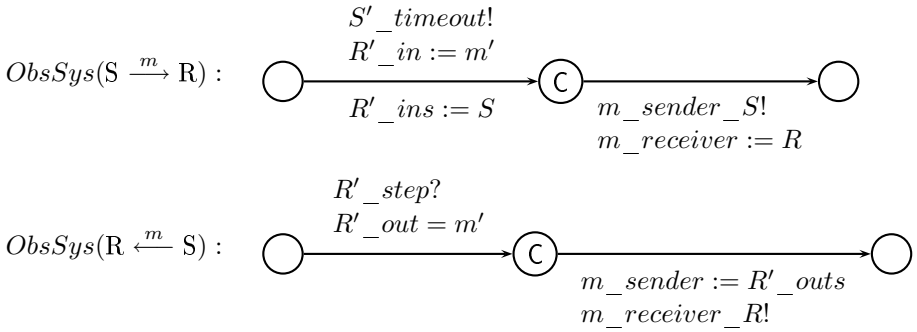


Abbildung 3.21: Observierbare asynchrone Kommunikation

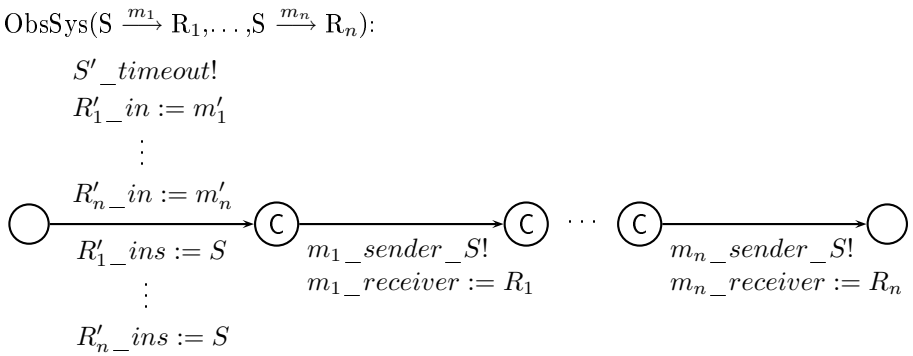


Abbildung 3.22: Observierung mehrerer Sendeereignisse

ximal ein Ereignis an eine Instanz geschickt werden darf (Größenbeschränkung der Eingabevariablen, vgl. 3.2) ist dieses unproblematisch, da so die Reihenfolge keine Rolle spielt.

In Abbildung 3.23 wird der Empfang eines Ereignisses dargestellt. Die Beschriftung ist komplementär zur Modifikation im System. Ist einem Ereignis eine Uhr zugeordnet, weil dessen Auftrittszeitpunkt Teil einer zeitlichen Bedingungen ist, muss bei seiner Beobachtung noch diese Uhr zurückgesetzt werden.

Überprüfung von Zeitbedingungen. Zeitbedingungen, die zulässig sind, wurden syntaktisch so beschränkt, dass sie sich direkt in Zeitautomaten als Transitionsbedingungen verwenden lassen. Jeder verwendeten Zeitmarke wurde eine Uhr zugeordnet, die zurückgesetzt wird, wenn das entsprechende

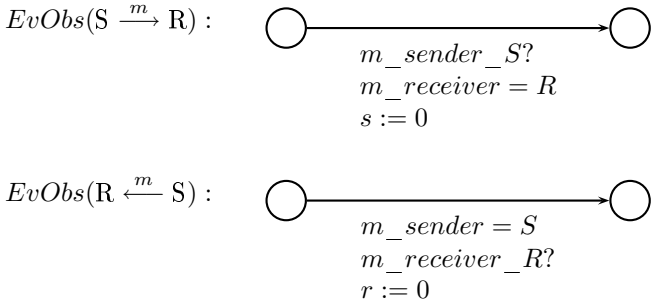


Abbildung 3.23: Observierung und Zurücksetzen von Uhren

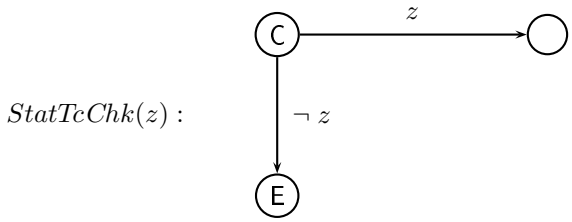


Abbildung 3.24: Statische Zeitbedingung

Ereignis eintritt. Für alle Zeitbedingungen mit nur einer Zeitmarke wird eine globale Uhr verwendet. Eine solche Bedingung schränkt das Auftreten des zugehörigen Ereignisses bezüglich der globalen Zeit ein.

Bei Zeitbedingungen mit zwei Zeitmarken ist zu beachten, dass die Uhr des früheren von zwei Ereignissen einen *größeren* Wert hat. Dieses entspricht nicht der intuitiven Zählweise, in der frühere Ereignisse kleinere Werte haben. Dieses wird in der Implementierung durch ein Vertauschen der Uhrenwerte ausgeglichen.

Kann eine Zeitbedingung direkt nach dem späteren Ereignis überprüft werden, wird eine zweite Uhr für das spätere Ereignis nicht benötigt, da deren Wert 0 ist. In diesem Fall wird diese redundante Uhr eingespart.[10]

Wir haben in Abschnitt 3.3.2.3 zwischen statischen und dynamischen Zeitbedingungen unterschieden. Statische Zeitbedingungen sind einfach zu überprüfen, da sich bereits zur Übersetzungszeit ermitteln lässt, an welcher Stelle in den Zeitautomaten die Überprüfung vorzunehmen ist. In Abbildung

[10] In der Implementierung können Uhren vielfach wiederverwendet werden. Auf derartige Optimierungen wird hier nicht weiter eingegangen.

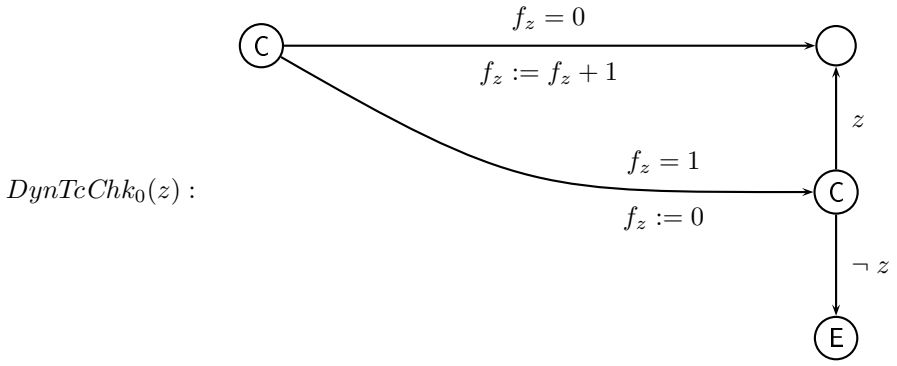


Abbildung 3.25: Dynamische Zeitbedingung

3.24 ist dargestellt, wie sich eine statische Zeitbedingung z direkt mit Transitionen realisieren lässt, die entweder den vorgesehenen Ablauf fortsetzen oder in einen Fehlerzustand verzweigen.

Die Behandlung von dynamischen Zeitbedingungen ist etwas aufwendiger, da erst zur Laufzeit festgestellt werden kann, welches der beteiligten Ereignisse später auftritt und wo die Überprüfung stattfinden muss. Daher wird pro dynamischer Zeitbedingung eine boolesche Variable f_z eingeführt, die signalisiert, ob *ein* Ereignis der Zeitbedingung bereits stattgefunden hat (Abb. 3.25). Diese Konstruktion wird bei beiden Ereignissen in den Observerautomaten eingefügt. Die oben abgebildete Transition behandelt den Fall, dass noch kein Ereignis der Zeitbedingung z stattgefunden hat. In diesem Fall wird lediglich der Wert von f_z verändert. Die untere Transition überprüft die Zeitbedingung, da f_z anzeigt, dass das andere beteiligte Ereignis bereits stattgefunden hat.

Im Zusammenhang mit Schleifen ergibt sich ein prinzipielles Problem, wenn zugelassen wird, dass sich die Instanzen in unterschiedlichen Iterationen„überholen“: Im allgemeinen Fall könnte eine Instanz unbeschränkt viele Sendeereignisse produzieren, die noch nicht empfangen wurden. Um in diesem Kontext Zeitbedingungen zu verifizieren, benötigte man unbeschränkt viele Uhren. Dieses ließe sich nicht mehr verifizieren. Daher wird gefordert, dass eine Nachricht empfangen werden muss, bevor sie in einer Schleife erneut versendet wird. In diesem Fall wird die Konstruktion wie in Abbildung 3.26 dargestellt abgeändert. Um zu verhindern, dass die zweite Wiederho-

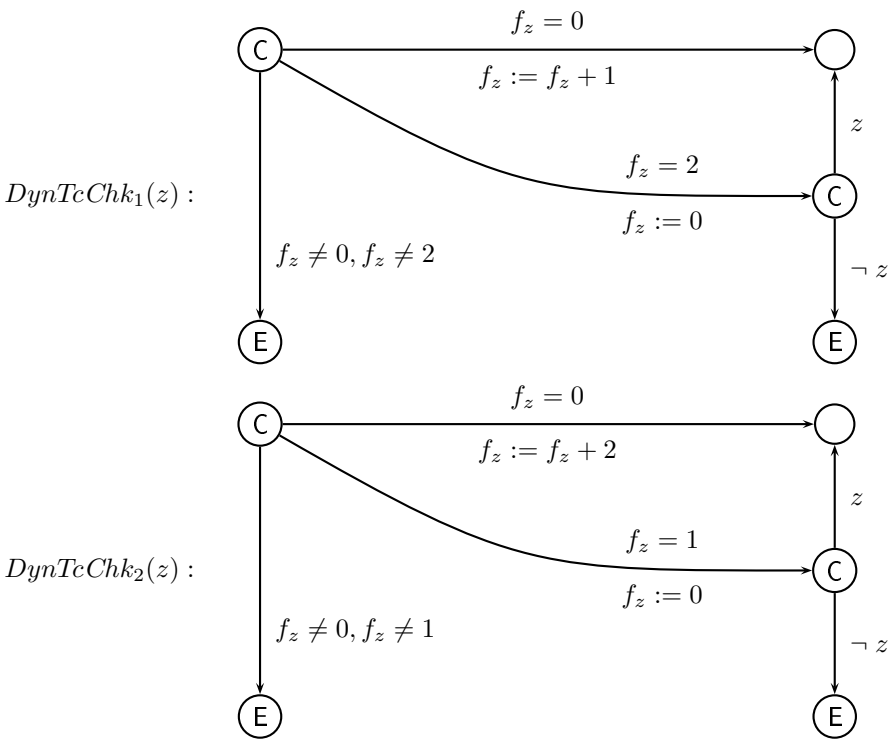
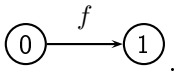


Abbildung 3.26: Dynamische Zeitbedingung in Schleifen

lung desselben Ereignisses zu einer Überprüfung statt zu einem Fehler führt, wird jedem beteiligten Ereignis ein spezifischer Wert zugeordnet. Wird festgestellt, dass f_z bereits den Wert des gerade beobachteten Ereignisses hat, ist ein unzulässiges Überholen von Iterationen aufgetreten und es wird in einen Fehlerzustand verzweigt (Fehlerzustand links unten). Ansonsten erfolgt die Überprüfung wie bisher.

Sind an einem Ereignis mehrere Zeitbedingungen zu überprüfen, wird dieses sequentiell durchgeführt, wobei die einzelnen Elemente durch *committed*-Zustände verbunden sind.

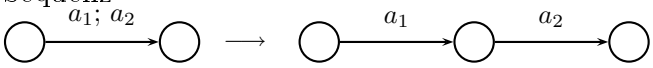
Konstruktion der Observer. Wir gehen davon aus, dass eine Ereignisfolge f gegeben ist, zu der ein Observer konstruiert werden soll. Dabei wird ein rekursives Verfahren angewendet. Wir beginnen mit



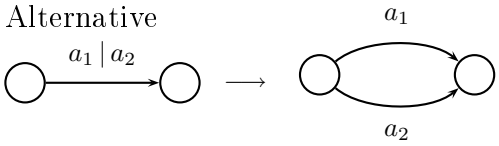
.

Je nach Aufbau von f wird der Observer sukzessive aufgebaut:

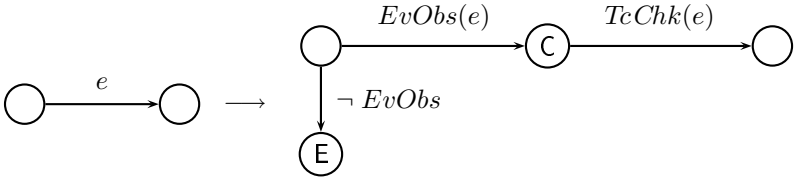
- Sequenz



- Alternative



- Einzelnes Ereignis (Konstruktion wie im Abschnitt zuvor erläutert)



- Schleifen (Beschreibung siehe unten)

Schleifen werden grundsätzlich wie in Abbildung 3.27 konstruiert. Die Zählvariable n der Schleife wird anfangs initialisiert und dann am Ende jeder

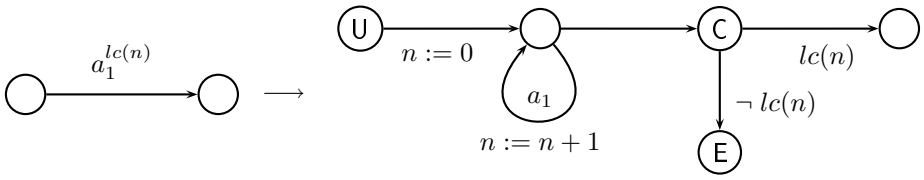


Abbildung 3.27: Schleifen-Observer

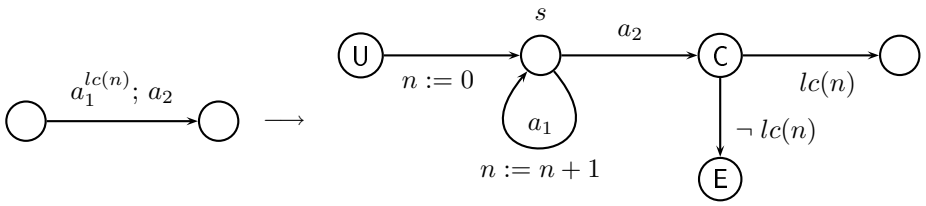


Abbildung 3.28: Schleifen-Observer mit deterministischem Ende

Iteration inkrementiert. Die Schleifenbedingung $lc(n)$ wird nach Beendigung der Schleife überprüft. Ein Problem dieser Konstruktion ist der Nichtdeterminismus, der beim Abbruch der Schleife entsteht. Um eine Schleife deterministisch abbrechen zu können, muss das erste Ereignis nach der Schleife in die Konstruktion mit einbezogen werden (siehe Abb. 3.28).

Die Schleifenbedingung kann nur überprüft werden, wenn a_2 eintritt. Daher können so prinzipiell nur Sequenzdiagramme verifiziert werden, die nicht mit einer Schleife enden, sondern noch wenigstens ein weiteres Ereignis enthalten. Der Model-Checker überprüft dann, ob dieses Ereignis stattfindet. Wenn nicht, liegt ein fehlerhafter Ablauf vor und die Schleifenbedingung braucht nicht mehr überprüft werden. Wenn das Ereignis auftritt, kann auch die Schleifenbedingung überprüft werden.

Auch in dieser Konstruktion kann Nichtdeterminismus auftreten, wenn das erste Ereignis nach der Schleife mit dem ersten Ereignis in der Schleife übereinstimmt. Wird die falsche Fortsetzung gewählt, werden im weiteren Ablauf Fehlerzustände betreten, obwohl ein System den Anforderungen entspricht. Da beim Model-Checking *alle* Ausführungspfade untersucht werden, werden diese nicht korrekten Fehlerpfade in jedem Fall aufgedeckt. Prinzipiell lässt sich dieser Nichtdeterminismus in vielen Fällen einfach durch Verschieben der Entscheidung über das Ende der Schleife hinaus auflösen (*delayed choice*): Sobald das erste Ereignis auftritt, das den Ablauf innerhalb der

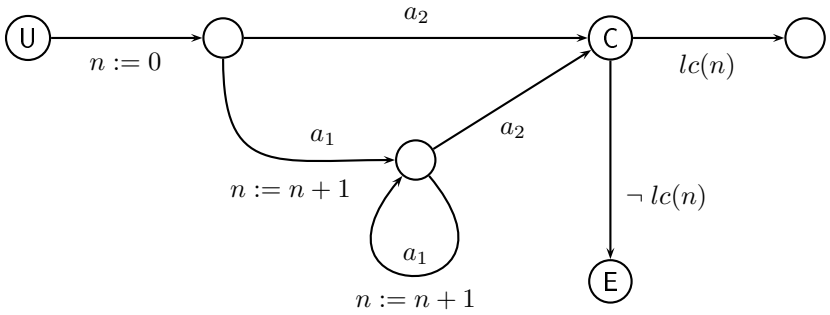


Abbildung 3.29: Schleifen-Observer mit zeitlicher Referenz auf die erste Iteration

Schleife von dem nachfolgenden Ablauf unterscheidet, kann entschieden werden, ob die Schleife verlassen wurde oder sich das System in der nächsten Schleifeniteration befindet.

Werden Schleifen geschachtelt, kann es schnell zu unübersichtlichen Konstruktionen kommen. Daher wird hier nur der Fall betrachtet, dass sich das Ende einer Schleife durch das erste Ereignis danach deterministisch ermitteln lässt. Für praktische Anwendungen trifft dieses in der Regel zu. Aus Abbildung 3.28 ergibt sich weiterhin die Einschränkung, dass Schleifen nicht am Ende einer Spezifikation stehen dürfen. In diesem Fall könnte ein Observer nicht das Ende eines Szenarios ermitteln.

Es kann notwendig sein, Zeitbedingungen zu formulieren, die verschiedene Iterationen einer Schleife umfassen (vgl. 2.1):

- Um den ersten Durchlauf einer Schleife referenzierbar zu machen, werden darin enthaltene Ereignisse mit e_{first} bezeichnet.
- Will man sich auf den letzten Durchlauf beziehen, lässt sich für derartige Ereignisse e_{last} verwenden.
- Um einen Zusammenhang zwischen einem aktuellen und dem kommenden Durchlauf herzustellen, wird für Ereignisse des nächsten Durchlaufs e_{next} verwendet.

Die Verwendung von Zeitbedingungen, die e_{first} enthalten, zieht eine Modifikation der Observerkonstruktion nach sich, da sich ansonsten Ereignisse der ersten Schleifeniteration nicht von Ereignissen anderer Wiederholungen

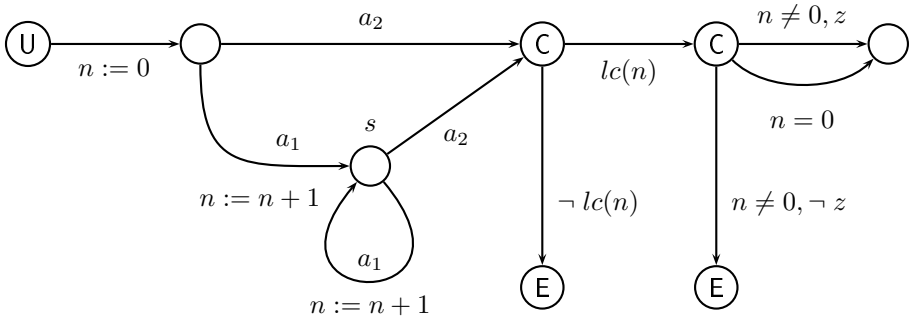


Abbildung 3.30: Schleifen-Observer mit zeitlicher Referenz auf die letzte Iteration

differenzieren lassen. Eine Möglichkeit ist das Aufrollen der ersten Schleifeniteration (Abb. 3.29).

Die Überprüfung von Zeitbedingungen, die ein Ereignis e_{last} des letzten Schleifendurchlaufs enthält, kann erst durchgeführt werden, wenn die Schleife als beendet erkannt wurde. Bezieht sich die Zeitbedingung auf einen Zusammenhang zwischen einem e_{last} und einem Ereignis nach der Schleife, findet die Überprüfung, wie oben beschrieben, am späteren Ereignis statt. Die Einführung einer zweiten Uhr kann entfallen. Im anderen Fall muss zunächst die Terminierung der Schleife abgewartet werden. In diesem Fall wird die Zeitbedingung z wie in Abbildung 3.30 aufgezeigt überprüft. Nachdem die Schleifenbedingung $lc(n)$ verifiziert wurde, wird z für eine Iterationszahl $n \neq 0$ kontrolliert. Für eine Iterationszahl $n = 0$ sind Zeitbedingungen, die e_{first}, e_{last} oder e_{next} enthalten, nicht definiert.

Zeitbedingungen, die sich mit e_{next} auf den kommenden Schleifendurchlauf beziehen, können wie gehabt überprüft werden. Es gilt allerdings die Einschränkung, dass das zeitlich spätere Ereignis e_{next} in der Schleife textuell *vor* dem Referenzereignis stehen muss. Ansonsten würde die Uhr des Referenzereignisses verfrüht zurückgesetzt. Da die Zuweisungen zwischen Uhren und ganzzahligen Variablen in UPPAAL eingeschränkt sind, lässt sich dieses Problem nicht durch einfache Konstruktionen beheben. Da dieses Konstrukt vor allem dazu gedacht ist, eine Verknüpfung des letzten Ereignisses einer aktuellen Iteration mit dem ersten Ereignis der nächsten Iteration herzustellen, reicht die vorgestellte Lösung in der Regel für praktische Problemstellungen aus.

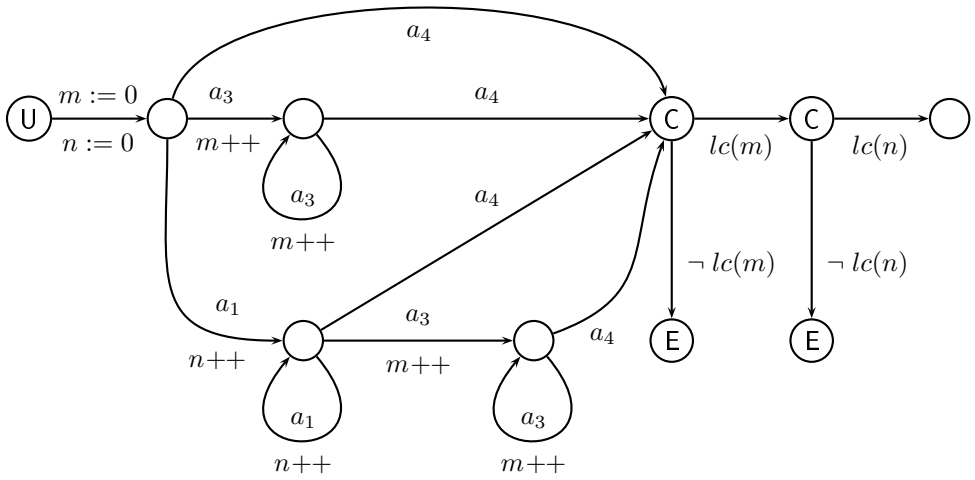


Abbildung 3.31: Geschachtelte Schleifen

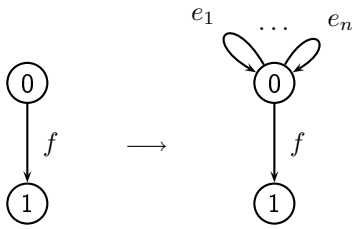


Abbildung 3.32: Ergänzung des Observers für optionales Verhalten

Bei Schleifen, die direkt aufeinander folgen, wird die Konstruktion so abgeändert, dass die Initialisierung der Schleifenzähler zusammen am Anfang erfolgt. Die Überprüfung der Schleifenbedingungen erfolgt ebenfalls zusammengefasst am Ende der Schleifensequenz. In Abbildung 3.31 ist dies für die Ereignisfolge $a_1^{lc(n)}, a_3^{lc(m)}, a_4$ dargestellt (a_1, a_3 und a_4 können wiederum Ereignissequenzen sein). Bei geschachtelten Schleifen muss die Initialisierung der Schleifenzähler ebenfalls vorgezogen werden. Die Überprüfung der Schleifenbedingung hingegen erfolgt wie im Normalfall einer einfachen Schleife.

Erreichbarkeit von Fehler- und Endzuständen. Bevor wir jetzt zu der Erzeugung der temporallogischen Formeln kommen, muss bei Observerautomaten, die optionales Verhalten beobachten, noch eine kleine Modifikation

vorgenommen werden. In Abbildung 3.32 ist zu sehen, wie der Startzustand eines Automaten, der die optionale Ereignisfolge f überwacht, um Transitionsschleifen ergänzt wird. Für jedes Ereignis e_i, mit dem die Sequenz begonnen werden kann, wird eine solche Schleife eingefügt. Damit wird sichergestellt, dass der Endzustand dieser Sequenz auch dann erreicht werden kann, wenn ein unvollständiger Präfix der Folge auftritt. Mit den Transitionsschleifen wird nichtdeterministisch „geraten“, ob ein erfüllender Systemlauf gerade beginnt. Da wir für optionales Verhalten nur fordern, dass wenigstens *ein* Ablauf des Systems das Sequenzdiagramm erfüllt, schadet es nichts, wenn ein unvollständiger Verhaltenspräfix in einem Fehlerzustand endet.

Weiterhin muss der Endzustand von obligatorischen Szenarien noch mit einer Transition zu einem Fehlerzustand E_{final} versehen werden. Diese Transition wird durch alle relevanten Ereignisse ausgelöst. Damit soll sichergestellt werden, dass ein Szenario mit Erreichen des Endzustandes beendet ist. Jedes nachfolgende relevante Ereignis wird als Fehler gewertet.

Ein Observer kann in vier Fällen eine Verletzung der Spezifikation feststellen. In diesen Fällen wird in zugehörige Fehlerzustände verzweigt:

- Es wird ein falsches Ereignis E_{event} beobachtet,
- ein Endzustand eines obligatorischen Szenarios wurde betreten und wieder verlassen (E_{final}),
- eine Zeitbedingung verletzt (E_{time}) oder
- eine Schleifenbedingung nicht erfüllt (E_{loop}).

Die Verifikation dieser Fehler reduziert sich nun auf die Erreichbarkeitsprüfung aller Fehlerzustände E_i mit $\forall\Box\, \neg E_i$. Für jeden Fehlerzustand eines obligatorischen Ablaufs wird eine entsprechende temporallogische Anforderung generiert. Ist ein Sequenzdiagramm als optional gekennzeichnet, wird die Erreichbarkeit der Fehlerzustände *nicht* geprüft, da in diesem Fall lediglich der aktuelle Durchlauf zurückgewiesen und nach einem anderen erfüllenden Lauf gesucht werden muss. Wird im obligatorischen Fall ein Fehlerzustand erreicht, wird ein Fehlerpfad generiert. Im Abschnitt 3.3.4 wird beschrieben, wie ein solcher Fehlerpfad auf die Ebene der UML projiziert werden kann.

Außer der Fehlerfreiheit muss die Terminierung eines Szenarios sichergestellt werden. Für obligatorisches Verhalten muss jeder Pfad im Endzustand G enden: $\forall\Box G$. Für optionales Verhalten soll es wenigstens einen erfüllenden

Durchlauf geben: $\exists\Diamond\, G$. Zusammenfassend werden folgende Anforderungen für ein Sequenzdiagramm erzeugt:

$$REQ = \begin{cases} \{\forall\Box\, G, \forall\Box\, \neg E_i | \textit{für alle Fehlerzustände } E_i\}, & \textit{falls mandatory} \\ \{\exists\Diamond\, G\}, & \textit{falls optional} \end{cases}$$

Weiterhin lässt die Implementierung auch die Eingabe benutzerdefinierter Anforderungen zu. Beispielsweise wird oft die Suche nach *Deadlocks* benötigt: $\forall\Box\, \neg deadlock$.

Beobachtung eines Systems vom Sequenzdiagrammen. Die Beobachtung eines Systems mittels einer *Menge* von Sequenzdiagrammen gestaltet sich umständlich, wenn nur bilaterale Kommunikation zwischen observierbar gemachtem System und den Observern möglich ist. Bilaterale Kommunikation war lange Zeit die einzige Form von Synchronisationsaktionen, die UPPAAL zuließ. Erst ab der Version 3.4 ist auch eine Synchronisation über Broadcast-Kanäle möglich. Da eine Broadcast-Aktion $a!$ in allen potentiellen Observern Transitionen, die mit $a?$ beschriftet sind, auslösen kann, lassen sich unterschiedliche Szenarien problemlos gleichzeitig beobachten.

Interne Kommunikation und nicht relevante Nachrichten. Bei der Beobachtung der Kommunikation zwischen den Instanzen soll oft interne Kommunikation *innerhalb* einer Komponente unberücksichtigt bleiben. Da es zulässig ist, einer Instanz *mehrere* Zustandsdiagramme zuzuordnen, kann diese Kommunikation ohne Weiteres auftreten.

Mittels eines Parameters kann in der Implementierung vom Modellierer bestimmt werden, ob interne Kommunikation für die Analyse relevant ist. Im asynchronen Modell können Sendeereignisse bereits zur Übersetzungszeit als intern identifiziert und ausgeblendet werden, indem sie einfach übergangen werden. Für Empfangsereignisse ist das nicht möglich, da zur Übersetzungszeit nicht bekannt ist, von welcher Zustandsmaschine ein Signal geschickt wird. Im synchronen Fall ist zur Übersetzungszeit weder für Empfangs- noch für Sendeereignisse ermittelbar, ob sie intern sind. Daher werden *alle* Ereignisse durchgängig beobachtbar gemacht. Allerdings werden alle Transitionen in Fehlerzustände E_{event} entfernt, die sich auf interne Kommunikation beziehen, d.h. das Auftreten interner Kommunikationsereignisse führt nicht in einen Fehlerzustand. Da eine Kommunikationsaktion über einen Broadcast-Kanal auf Senderseite nicht blockiert, ist dieses Verfahren unkritisch.

Zusätzlich kann für ein Sequenzdiagramm eine Menge nicht relevanter Nachrichten benannt werden. In diesem Fall werden ebenfalls alle Transitionen entfernt, die sich auf Ereignisse derartiger Nachrichten beziehen und in Fehlerzustände führen.

Insgesamt erhalten wir als Ergebnis der Übersetzung folgendes System von Zeitautomaten

- die observierbar gemachten Automaten,
- die Observerautomaten und

sowie im Falle asynchroner Kommunikation zusätzlich noch

- den Schritt- und den Kontrollautomaten und
- die modifizierten Ereigniswarteschlangen.

Mit Hilfe von generierten Verifikationsanfragen kann eine automatische Prüfung der Konsistenz von Sequenz- und Zustandsdiagrammen durchgeführt werden. Es stellt sich die Frage, was passiert, wenn ein Fehler ermittelt wird. Zur Fehleranalyse sollten dem Modellierer die Informationen des Fehlerpfades in lesbarer Form angeboten werden. Damit beschäftigt sich der nächste Abschnitt.

3.3.4 Visualisierung eines Fehlerpfades

Die Verifikation mit Hilfe eines Model-Checkers liefert einen Fehlerpfad. Leider ist dieser in seiner rohen Form zunächst kaum hilfreich, da er sich auf das Ergebnis der Transformation von UML-Modellen in Zeitautomaten bezieht und nur mit Expertenwissen über deren Konstruktion deuten lässt. Diese Transformation sollte aber für den Anwender transparent bleiben. Daher ist eine Rücktransformation auf UML-Ebene notwendig.

Für eine Verifikation werden sowohl die Menge von Zustandsdiagrammen, als auch die Menge der Sequenzdiagramme in ein Automatensystem abgebildet. Für eine Visualisierung kommen daher prinzipiell zwei Möglichkeiten in Betracht:

- Der Fehlerpfad wird in einem Sequenzdiagramm angezeigt. Es sollte die Position im Sequenzdiagramm markiert werden, an der eine Verletzung des vorgesehenen Ablaufs aufgetreten ist. Zusätzlich wären Informationen wünschenswert, die bei einer Fehlerdiagnose weiterhelfen. Beispielsweise sollten aktuelle Variableninhalte ausgegeben werden. Eine intuitive Darstellungsform ist ein UML-Kommentar, der an die Stelle im Diagramm geheftet wird, an der ein abweichendes Verhalten seinen Anfang nimmt.

- Eine Visualisierung des fehlerhaften Ablaufs wäre ebenfalls möglich, indem Zustände auf dem Niveau der Zeitautomaten mit Zuständen in den Zustandsdiagrammen verbunden werden. In diesem Fall könnte der Ablauf mit Hilfe einer Animation darstellt werden, in der jeweils aktive Zustände bis zum Auftreten einer fehlerhaften Konfiguration farbig hervorgehoben werden.

Wir haben hier den ersten Ansatz verfolgt, da sich auf diese Weise fehlerhaftes Verhalten übersichtlich und kompakt darstellen lässt. Außerdem lässt sich so die Fehlerausgabe auch für UML-Werkzeuge realisieren, die eine Simulation nicht unterstützen.

Ein Fehlerpfad in UPPAAL enthält Konfigurationen von Zeitautomaten einschließlich eines Zustandsvektors, in dem jeder Prozess repräsentiert wird, und aktuelle Variablenwerte sowie Differenzen von Uhrenwerten. Weiterhin lassen sich die Transitionen entnehmen, die jeweils geschaltet haben. Bei der Interpretation des fehlerhaften Ablaufs stellt sich das Problem, dass im Fehlerpfad Automatenzustände und Übergänge, die sich auf Zustandsdiagramme beziehen, durchmischt sind mit denen, die für die Sequenzdiagramme relevant sind. Da zusätzlich viele Teile des Fehlerpfades sich auf Hilfskonstruktionen beziehen, ist ein großer Teil für eine Visualisierung uninteressant. Daher muss ein Fehlerpfad zunächst auf die relevanten Teile gefiltert werden. Auf der Ebene von Zeitautomaten wird dafür ein genaues Wissen über die verwendete Transformation benötigt, um die Bedeutung von Automatenzuständen und Transitionen für die UML-Ebene zu erkennen.

Dieses lässt sich vereinfachen, wenn bereits bei der Abbildung die relevanten Teile in den Zeitautomaten per Namenskonvention gekennzeichnet werden (Abb. 3.33). Eine besonders wichtige Rolle spielen dabei Zustände in einem Sequenzdiagramm, in denen Zeit vergehen kann. Davon gibt es jeweils einen zwischen zwei Ereignissen (Senden oder Empfangen einer Nachricht)

an der Lebenslinie einer Instanz. Werden diese Zustände bei der Abbildung in Zeitautomaten mit dem Diagrammnamen, der zugehörigen Instanz und der genauen Position innerhalb des Diagramms beschriftet, findet sich diese Information später leicht im Fehlerpfad wieder und kann dann in das betroffene Sequenzdiagramm eingeblendet werden.

In ähnlicher Weise gilt es auch, die Fehlerzustände e_i zu markieren. Diese sollten insbesondere per Namenskonvention einen Hinweis auf die Art des aufgetretenen Fehlers geben (Sequenzfehler, zeitliche Verletzung einer Anforderung oder Iterationsfehler).

Nach einer derartigen Vorbereitung gestaltet sich die Rückabbildung des Fehlers in ein Sequenzdiagramm relativ einfach: Da auf Erreichbarkeit eines Fehlerzustandes geprüft wird, befindet sich ein solcher Zustand am Ende des Fehlerpfades. Verfolgt man von dort aus den Pfad rückwärts bis zum ersten Auftreten eines markierten Sequenzdiagrammzustandes, hat man die Stelle im Diagramm identifiziert, an der ein Fehler auftritt. Ein Sequenzfehler und eine Verletzung einer zeitlichen Bedingung sind dabei dem Ereignis zuzuordnen, das im Sequenzdiagramm dem markierten Zustand folgt. Ein Iterationsfehler sollte der vorangegangenen Schleifenkonstruktion zugeschrieben werden, da derartige Fehler erst nach Verlassen einer Schleife aufgedeckt werden können.

Weitere Informationen über die aktuelle Konfiguration lassen sich ebenfalls dem Fehlerpfad entnehmen. Verfolgt man den Pfad weiter rückwärts und entnimmt jeweils die Konfiguration bei den markierten Zuständen, lässt sich so die gesamte Vorgeschichte, die zum Fehler führt, rekonstruieren.

3.4 Verifikation von QNX-spezifischen Modellen

Die Modellierung von plattformspezifischen Elementen von QNX wurde in Abschnitt 2.3 beschrieben. Dort werden die Grundlagen für die folgende Analyse gelegt. Ausgangspunkt dieser Transformation sind zwei Sichtweisen auf eine QNX-Anwendung: Ein *Architekturmodell* enthält eine Übersicht über Prozesse mit zugeordneten Threads und Kommunikationskanälen. Dieses Modell entspricht einem UML-Objektdiagramm. Ein *Dynamikmodell* beschreibt für jeden Thread dessen Verhalten in Form von UML-Zustandsdiagrammen.

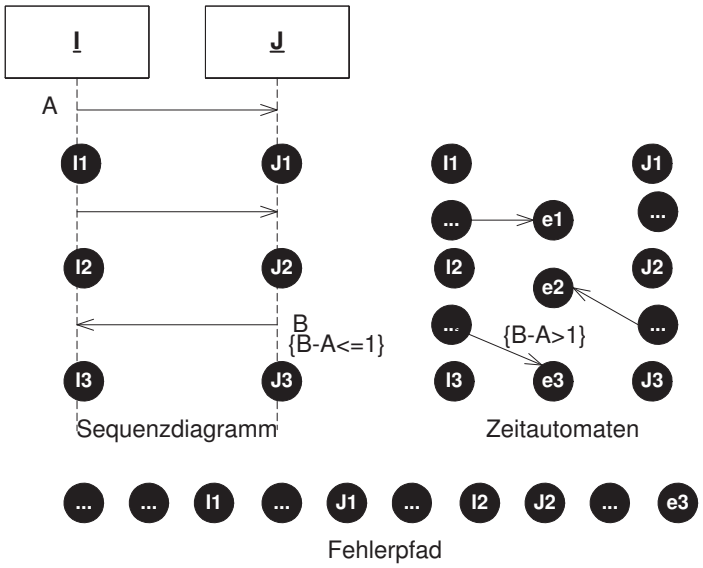


Abbildung 3.33: Prinzip der Visualisierung eines Fehlerpfades

Diese enthalten Zustände mit Ausführungszeiten von Aktionen und Transitionen mit Kommunikationsaktionen. Daraus wird eine Darstellung in Form von Zeitautomaten generiert. Diese umfasst folgende Typen von Prozessen:

- Eine Menge von Thread-Automaten TA_{thread}. Diese modellieren das interne Verhalten von Threads.

- Eine Menge von Scheduler-Automaten $TA_{scheduler}$. Sie modellieren einen Scheduler, der für die prioritätengerechte Verarbeitung von Threads verantwortlich ist.

- Eine Menge von Kanal-Automaten $TA_{channel}$. Sie verwalten Warteschlangen, die den Kanälen zwischen Prozessen zugeordnet sind.

Die letzten zwei Automaten werden aus der Anwendungsarchitektur abgeleitet. Das Architekturmodell dient außerdem der Überprüfung, ob die Kommunikation zwischen Threads ausschließlich über Kanäle stattfindet. Die interne Beschreibung von Threads wird aus den ihnen zugeordneten Zustandsdiagrammen ermittelt.

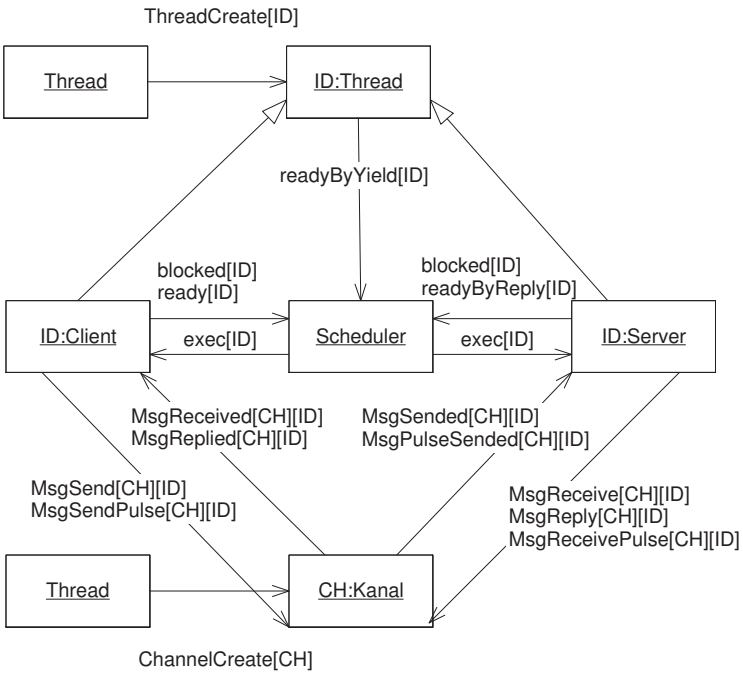


Abbildung 3.34: Übersicht über Kommunikationsaktionen

Die Kommunikation zwischen den Automaten findet über synchrone Aktionen statt. Abbildung 3.34 gibt einen Überblick über die benutzten synchronen Aktionen. Dabei werden Threads in ihren Rollen als Client und Server explizit dargestellt. Zu beachten ist insbesondere, dass Nachrichten niemals direkt zwischen zwei Threads weitergeleitet, sondern immer über eine Instanz eines Kanalautomaten vermittelt werden.

Im Folgenden beschreiben wir die Abbildung dieser Modelle in Zeitautomaten. Dabei werden als Konvention globale Variablen mit dem Präfix G und lokale Variablen mit L versehen. Konstanten werden mit Großbuchstaben benannt und Uhren beginnen mit Kleinbuchstaben.

3.4.1 Transformation von Zuständen

Grundsätzlich kann der Zeitverbrauch einer Aktion entweder von einer Scheduler-Komponente oder von den Thread-Automaten verwaltet werden. Wir folgen hier dem Ansatz aus [AGS00], in dem die Zeitverwaltung in den Thread-Komponenten realisiert wird. Dazu wird jeder UML-Zustand S eines Threads auf drei Automatenzustände *Ready*, *Running* und *Preempted* abgebildet (siehe Abb. 3.35). Diese Zustände repräsentieren den Status eines Threads bezüglich der Prozessornutzung. Der Hilfszustand *Init* ermöglicht ein Betreten eines Zustands, wenn zuvor keine Scheduling-Entscheidung notwendig war. Im Beispiel wird eine Ausführungszeit von $t_{exec} \in [T1, T2]$ modelliert.

Bei der Abbildung in Zeitautomaten werden folgende Variablen, Konstanten und Synchronisationsaktionen verwendet:

- ID ist eine eindeutige Identifizierung des modellierten Threads. Sie wird als Konstante realisiert.
- Die synchrone Aktion *exec* wird mit der Thread-Identifizierung parametrisiert. Sie erlaubt es dem Scheduler, einem Thread Rechenzeit zuzuweisen.
- Die synchrone Aktion *preempt* unterbricht den aktuell aktiven Thread.
- Die lokale Uhr *execTime* misst die Laufzeit eines Threads. Sie wird beim Betreten von *Running* zurückgesetzt.
- Die lokalen Variablen *Lmin* und *Lmax* enthalten eine obere und untere Grenze für die Restlaufzeit eines Threads. Aufgrund von Interferenzen

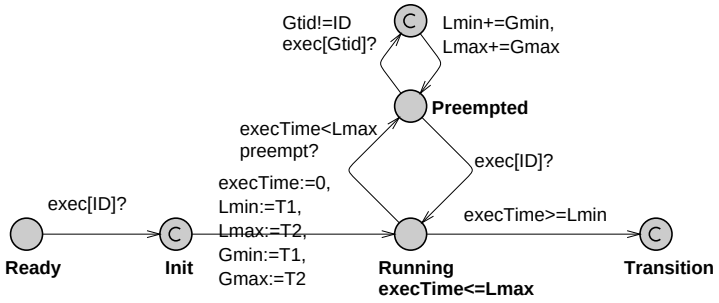


Abbildung 3.35: Transformation von Zuständen

mit hoch priorisierten Threads können diese Grenzen um die Unterbrechungszeit anwachsen. Da diese Unterbrechungszeiten durch Intervalle mit ganzzahligen konstanten Intervallgrenzen beschrieben werden, können ganzzahlige Variablen verwendet werden.

Im Zustand *Ready* wartet der Thread auf die Zuteilung von Rechenzeit durch den Scheduler. Dieses wird über die synchrone Aktion $exec[ID]$ signalisiert. Es findet ein Wechsel nach *Running* statt. Dort verbleibt die Kontrolle eines Threads, solange bis eine Aktion abgearbeitet ist oder der Thread unterbrochen wird. Die minimale und maximale Ausführungszeit wird mit der Invarianten $execTime <= Lmax$ von *Running* und der Bedingung $execTime >= Lmin$ an der rechten Transition sichergestellt.

Wenn der Scheduler die Ausführung vor Ende der maximalen Bearbeitungszeit mit *preempt* unterbricht, wechselt der Thread in den Zustand *Preempted*. Ein Thread kann dann entweder die Kontrolle sofort wieder zurück erhalten oder es werden im Falle der Aktivierung eines Threads mit höherer Priorität $Lmin$ und $Lmax$ um die minimale und maximale Unterbrechungszeit erhöht (Transitionsschleife am Zustand *Preempted*).

Diese Behandlung von Unterbrechungszeiten impliziert, dass ein Thread B mit höherer Priorität seine Aktion beendet, bevor der unterbrochene Thread A die Kontrolle erneut erlangt. Dieses trifft zu, da es an dieser Stelle nicht zu einer Umkehr im Prioritätenschema kommen kann: Ein hoch priorisierter Prozess kann z.B. durch einen Zugriff auf einen nicht verfügbaren Server blockiert werden. Dieses setzt aber eine Kommunikationsaktion voraus. Auf UML-Ebene befinden sich alle Kommunikationsaktionen an den Transitionen, so dass eine Blockierung *innerhalb* eines Zustands nicht auftreten kann.

Ein Thread B mit hoher Priorität ist erst *nach* seiner aktuellen Aktion durch einen Ressourcenzugriff blockierbar. Sehr wohl könnte es aber zu einer Prioritätsinversion bezogen auf den Gesamtablauf kommen, soweit sie nicht durch Prioritätenvererbung abgefangen wird. Weiterhin kann selbstverständlich der Thread B seinerseits durch einen Thread C mit noch höherer Priorität unterbrochen werden, usw. In diesem Fall kann A erst weiter verarbeitet werden, wenn B und C ihre Aktionen beendet haben. Die Transitionsschleife wird entsprechend oft durchlaufen.

Technisch gesehen wird durch Anpassung der minimalen und maximalen Restlaufzeiten *Lmin* und *Lmax* und deren Verwendung in Invarianten und Transitionsbedingungen eine einfache Form eines Systems mit $StopWatches$ realisiert. Es handelt sich dabei um eine sehr einfache Form, in der eine Uhr um einen *konstanten Betrag* angehalten wird. Dieses ist die einzige Möglichkeit, die UPPAAL-Zeitautomaten erlauben. Glücklicherweise ist diese Variante der simulierten $StopWatches$ für diesen Kontext absolut ausreichend. Im allgemeinen Fall führen anhaltbare Uhren zu unentscheidbaren Problemen.

3.4.2 Transformation von Transitionen

Als nächstes werden Transitionen in Zeitautomaten übersetzt. Die Darstellung hängt im Wesentlichen von den QNX-Kommunikationsaktionen ab, mit denen sie beschriftet sind. Diese Beschriftung auf UML-Ebene wird jeweils mit angegeben. Die Umsetzung in Zeitautomaten erfordert zahlreiche Synchronisationen mit den Kanal- und Scheduler-Automaten, die weiter unten eingeführt werden. Eine direkte Kommunikation zwischen Threads findet nicht statt, sondern sie wird jeweils über die Kanal-Automaten abgewickelt. Die resultierenden Automaten-Fragmente für Transitionen verbinden jeweils die Abbildungen der Zustände.

SendMessage(). Eine mit $/SendMessage\,(CH, ABC)$ beschriftete Transition wird wie in Abbildung 3.36 dargestellt in Zeitautomaten realisiert. Ein $MsgSend$ wird von einem Client ausgelöst. Wie beim QNX-Message-Passing üblich, folgen darauf ein $MsgReceived$ und ein $MsgReplied$, die vom Server initiiert werden. Dieses wird direkt in synchrone Aktionen auf Ebene der Zeitautomaten umgesetzt. Dabei werden der Kommunikationskanal CH und die eigene Thread-Kennung ID als Parameter übergeben. Die Aktionen $blocked\,[ID]!$ und $ready\,[ID]!$ sind Anweisungen an den Scheduler, den

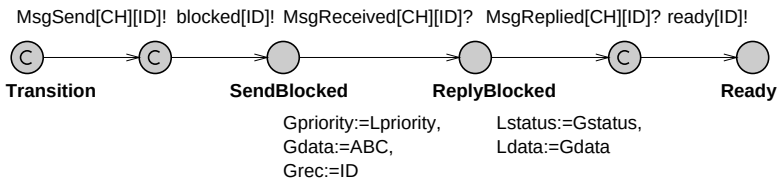


Abbildung 3.36: Transformation von SendMessage()-Transitionen

Thread ID aufgrund der vorübergehenden Blockierung aus der Menge der zur Verarbeitung bereiten Threads zu entfernen bzw. wieder einzufügen.

Mit $Gpriority := Lpriority$ wird die Priorität des Clients dem Server übergeben. Dieses bereitet eine Prioritätsvererbung zur Vermeidung einer Prioritätsinversion vor. $Gdata$ und $Gstatus$ sind globale Variablen, über die Daten ABC und ein Status an lokale Kopien $Ldata$ und $Lstatus$ übertragen werden.

ReceiveMessage(). Eine Transition eines Servers zum Empfangen von Nachrichten ist mit $/ReceiveMessage\,(CH)$ beschriftet. In Zeitautomaten wird entsprechend dem Server-Kommunikationszyklus zunächst der Kanal CH mit $MsgReceive$ darüber informiert, dass der Thread ID zum Empfang einer Nachricht bereit ist (Abb. 3.37). Analog zu oben wird der Scheduler über $blocked\,[ID]!$ und $ready\,[ID]!$ über den Zustand des Threads informiert. Eine verfügbare Nachricht wird dem Server vom Kanal-Automaten über die Aktion $MsgSended$ angekündigt. Dabei werden über die Variablen $Gdata$, $Gpriority$ und $Grec$ Daten, Client-Priorität und Absender übertragen.

Der Kanal-Automat sorgt dafür, dass ein $MsgSended$ direkt im Anschluss an ein $MsgReceived$ stattfindet, so dass die globalen Variablen nicht überschrieben werden. Die Client-Priorität wird übermittelt, um die Vererbung von Prioritäten zu realisieren. Sie wird in eine globale Prioritätentabelle eingetragen. Dieses entspricht der Implementierung unter QNX und vermeidet in vielen (allerdings nicht allen) Fällen die Umkehr der Prioritätenordnung aufgrund von Server-Aufrufen.

ReplyMessage(). Ein Reply $/ReplyMessage(CH, status, ABC)$ wird wie in Abbildung 3.38 umgesetzt. Mit einem $MsgReply$ werden eine Statusmel-

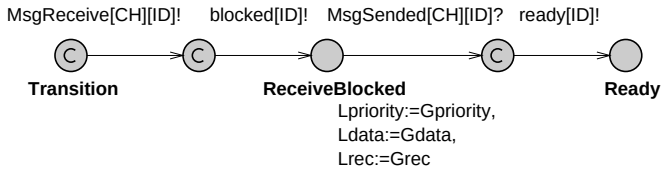


Abbildung 3.37: Transformation von ReceiveMessage()-Transitionen

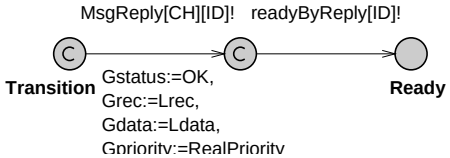


Abbildung 3.38: Transformation von ReplyMessage()-Transitionen

dung $Gstatus$, der Empfänger $Grec$ und Daten $Gdata$ gesetzt. Für den Scheduler wird die ursprüngliche Priorität übergeben, die vorübergehend durch eine vererbte Priorität ersetzt wurde. Da sich daher eine Veränderung im Scheduling ergeben kann, wird über $ReadyByReply$ eine Kommunikation mit dem Scheduler durchgeführt, obwohl ein $ReplyMessage$ selbst nicht blockiert. Der Scheduler entfernt den Thread ID aus der Warteschlange der geerbten Priorität, stellt die Orginalpriorität wieder her und fügt ihn in die Warteschlange der aktualisierten Priorität ein.

SendPulse(). Mit $/SendPulse\,(CH)$ wird ein Puls an einen Kanal CH abgegeben (Abb. 3.39). Kennzeichnend für einen Puls ist, dass der Sender nicht blockiert. Über $MsgSendPulse$ wird die Nachricht an den Kanal CH übergeben. Eine neue Scheduling-Entscheidung ist nicht notwendig, da weder eine Blockierung auftritt, noch die Priorität des Threads verändert wird. Daher führt diese Transition direkt zum folgenden Zustand.

ReceivePulse(). Ein Puls wird an einer Transition ähnlich wie eine Nachricht empfangen ($/ReceivePulse\,(CH)$). Auf Ebene der Zeitautomaten wird dieses analog zu $ReceiveMessage$ realisiert. Es werden allerdings weder Variablen zur Prioritäts- und Datenkopierung benötigt, noch muss der Absender gespeichert werden (Abb. 3.40).

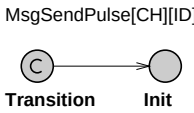


Abbildung 3.39: Transformation von SendPulse()-Transitionen

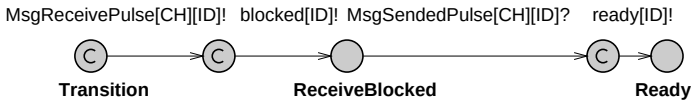


Abbildung 3.40: Transformation von ReceivePulse()-Transitionen

Yield(). Die Freigabe eines Prozessors mit $Yield()$ erfordert lediglich eine neue Scheduling-Entscheidung (Abb. 3.41). Der Scheduler verschiebt einen Thread dabei an das Ende der Warteschlange.

Sleep(). Eine *Sleep*-Anweisung bewirkt, dass sich der Thread vorübergehend mit *blocked* beim Scheduler aus der Menge der rechenwilligen Threads abmeldet. Nach Ablauf der Zeitspanne $TIME$ meldet sich der Thread wieder beim Scheduler zurück (Abb. 3.42).

Create-Anweisungen. Es können sowohl Prozesse als auch Threads erzeugt werden. Eine $CreateProcess(ID)$-Aktion wird dabei automatisch in einen $CreateThread(ID)$-Befehl umgewandelt, wobei als Parameter die ID des Main-Threads des Prozesses übergeben wird. In Abbildung 3.43 ist die

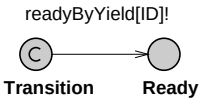


Abbildung 3.41: Transformation einer Yield()-Transition

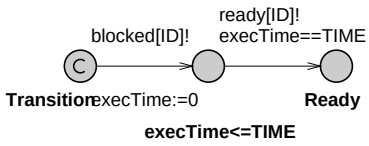


Abbildung 3.42: Transformation einer Sleep()-Transition

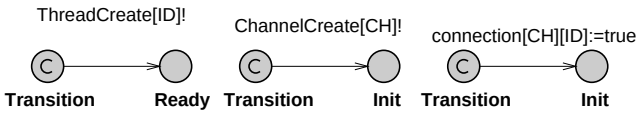


Abbildung 3.43: Transformation von Create-Transitionen

Umsetzung von verschiedenen Create-Aktionen dargestellt. Die Umsetzung erfolgt im Thread- bzw. im Kanal-Automaten.

- $/CreateThread\,(ID)$.
  Links: Die Erzeugung eines Threads ID wird mit $ThreadCreate\,[ID]!$ aus einem bereits aktiven Thread eingeleitet. Der Thread wird damit aus einem Pseudo-Zustand in seinen Anfangszustand überführt und meldet sich mit $ready\,[ID]!$ beim Scheduler an.

- $/CreateChannel\,(CH)$.
  Mitte: Der Kanal wird mit $ChannelCreate\,[CH]!$ erzeugt. Auch hier findet eine Transition von einem Pseudo-Zustand in den Startzustand $IDLE$ statt.

- $/AttachConnection\,(CH)$.
  Rechts: Es wird das Feld *connection* entsprechend gesetzt. Dieses Feld signalisiert, ob ein Thread Zugriff auf einen Kanal hat.

Transitionen ohne Kommunikationsaktion. Es gibt neben den bisher beschriebenen Transitionen auch Zustandsübergänge, die nicht mit einer besonderen Kommunikationsaktion beschriftet sind. In diesem Fall kann direkt zum Folgezustand gesprungen werden. Eine neue Scheduling-Entscheidung fällt nicht an.

3.4.3 Transformation sonstiger Konstrukte

Start- und Endzustände. Start- und Endzustände z_0 bzw. z_f werden wie normale Zustände transformiert. Es ist aber darauf zu achten, dass beim Systemstart lauffähige Threads in die Warteschlangen ihrer Priorität eingefügt werden. Dieses wird durch eine entsprechende Initialisierung desjenigen Feldes erreicht, das die Prioritäten-Warteschlange repräsentiert. Wenn ein Thread dynamisch erzeugt wird, enthält er eine Transition, die vom Initialzustand ausgeht und mit $ThreadCreate[ID]?$ beschriftet ist.

Für Endzustände z_f wird eine Transition zu einem Zustand z_{Final} hinzugefügt, der nicht mehr verlassen werden kann. Wird diese Transition genommen, wechselt der Thread mit $blocked\,[ID]$ in einen blockierten Zustand.

Deadlines von Tasks. Tasks können innerhalb des Zustandsdiagramms frei definiert werden (vgl. Abschnitt2.3). Mit diesem Konzept wird der zeitliche Abstand zwischen einer Transition mit einem Trigger-Stereotyp und einer Transition mit einem Response-Stereotyp mit einer vorgegebenen Deadline verglichen.

Zur Vorbereitung der Verifikation dieser Anforderung wird am Trigger eines Tasks t eine Uhr clk_t eingeführt und mit 0 initialisiert. An der als Response markierten Transition wird bei Einhaltung der Deadline ($clk\_t \leq deadline\,(t)$) der Ablauf fortgesetzt. Andernfalls wird in einen Fehlerzustand $Error\_t$ verzweigt. Dieser Fehlerzustand kann bei einer Verifikation auf Erreichbarkeit überprüft werden.

Wird ein Zustandsübergang, der als Response gekennzeichnet ist, bei der Transformation auf eine Folge von Transitionen abgebildet (z.B. bei einem $SendMessage()$), findet die Überprüfung nach der letzten regulären Transition statt.

Bedingungen und Variablen-Zuweisungen. Bedingungen und Zuweisungen an Variablen an UML-Zustandsübergängen werden unverändert übernommen werden, weil vorausgesetzt wurde, dass sie in UPPAAL-Syntax erfolgen. Wird ein Übergang auf eine Folge von Transitionen abgebildet, werden die Bedingungen und Zuweisungen der ersten Transition zugeordnet.

3.4.4 Modellierung eines Schedulers

Ein Hardware-Knoten enthält einen Scheduler (Abb. 3.44). Ein Scheduler wird durch synchrone Aktionen immer dann gestartet, wenn eine Scheduling-Entscheidung nötig ist. Eine anstehende Scheduling-Entscheidung kann verschiedene Ursachen haben, die zu unterschiedlichen Aktionen führen.

Um die Darstellung zu vereinfachen, werden alle Operationen auf Datenstrukturen in der Abbildung nur angedeutet. Es handelt sich dabei um einfache Listenoperationen, die auf UPPAAL-Ebene direkt zu realisieren sind. Jede Prioritätsstufe verfügt über eine Warteschlange der rechenwilligen Threads. Mit $removeFromQueues[x]$ wird ein Thread x aus der Warteschlange seiner Priorität entfernt. Eingefügt wird ein Thread mit $insertIn\ Queues[x]$. Zur Ausführung gelangt derjenige Thread, der in der Warteschlange mit der höchsten Priorität an erster Stelle steht. Dieser wird mit $searchFor\ Highest$ ermittelt, indem alle Warteschlangen beginnend mit der höchsten Priorität bis zum ersten nicht leeren Element durchlaufen werden.

Auf folgende Aktionen muss der Scheduler reagieren:

- $ready[x]$: Wird ein Thread aktiviert oder neu erzeugt, wird er in die entsprechende Warteschlange für eine bestimmte Prioritätsstufe eingefügt. Über die Aktion *preempt* wird der laufende Thread unterbrochen.

- $blocked[x]$: Blockiert ein Thread, wird dieser aus der Prioritätenwarteschlange gelöscht (linker Pfad).

- $readyByYield[x]$: Der Thread gibt freiwillig die Kontrolle ab, ist aber weiter rechenwillig. Daher wird er aus der Warteschlange entfernt und am Ende wieder eingereiht.

- $readyByReply[x]$: Es handelt sich in diesem Fall um einen Server. Aufgrund des *Priority Inheritence Protocols* wurde ein solcher Thread vorübergehend auf die Priorität des Clients angehoben. Dieses muss wieder rückgängig gemacht und der Thread aus der Warteschlange der Client-Priorität entfernt werden. Er wird stattdessen in die Warteschlange seiner Basispriorität eingefügt.

Nachdem alle nötigen Aktualisierungen der Prioritätswarteschlangen vorgenommen wurden, wird der höchst priorisierte Thread ermittelt und seine Ausführung mit $exec[ID]!$ bewirkt.

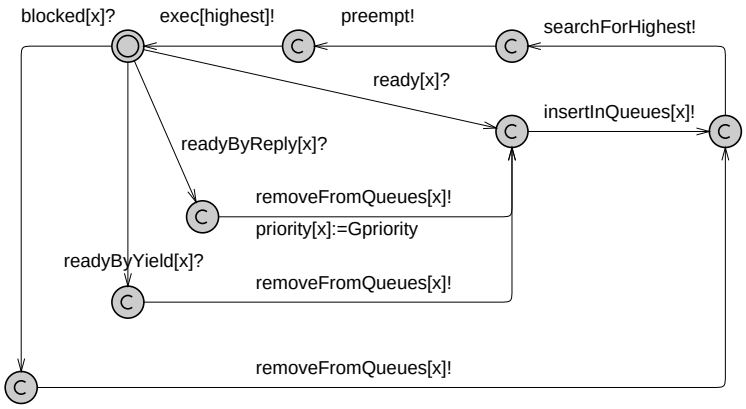


Abbildung 3.44: Modellierung eines Schedulers

3.4.5 Modellierung eines Kanals

Ein Kanal verbindet Clients und Server. Zugriffe verwaltet er in drei Warteschlangen: Die Warteschlange *SendBlocked* enthält durch ein Senden blockierte Clients, entsprechend die Warteschlange *ReceiveBlocked* blockierte Server und die Warteschlange *ReplyBlocked* Clients, deren Anfragen gerade von einem Server bearbeitet werden. Mit Hilfe der letzten Datenstruktur können Antworten dem anfragenden Client zugeordnet werden. Aus Effizienzgründen kann auf diese Datenstruktur verzichtet werden, wenn die ID des Clients dem Server mit übergeben wird. Daher wird im Folgenden auf eine explizite Modellierung von *ReplyBlocked* verzichtet.

Der Aufbau eines Automaten zur Verwaltung eines Kanals ist in Abbildung 3.45 und 3.46 beschrieben. Der Kanal-Automat wurde aus Gründen der Übersicht in fünf Bestandteile zerlegt. Er ist als Summe dieser Teile implementiert, wobei gleich benannte Zustände übereinstimmen. Jeder Bestandteil beschreibt die Reaktion auf eine bestimmte Kommunikationsanfrage. Datenoperationen auf den Warteschlangen werden nur angedeutet. Eine Implementierung einer Warteschlange in UPPAAL-Zeitautomaten wurde bereits in Abschnitt 3.2.1.3 vorgestellt. Hier kommt eine ähnliche Konstruktion zur Anwendung. Nachrichten und Pulse werden jeweils durch eine Konstante repräsentiert, wobei Nachrichten positive und Pulse negative Werte erhalten.

Ein Kanal wird durch ein $ChannelCreate[CH]!$ aktiviert (erzeugt) und befindet sich dann im Zustand *Idle*. Bevor ein Kanal von einem Thread

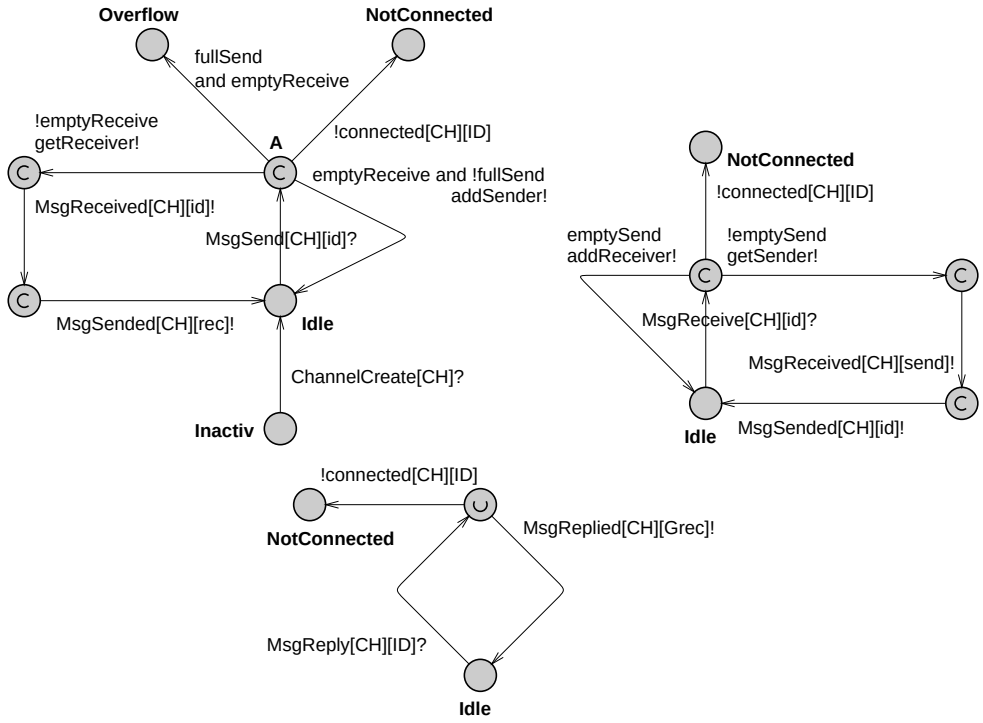


Abbildung 3.45: Modellierung eines Kanals (Senden, Empfangen, Beantworten)

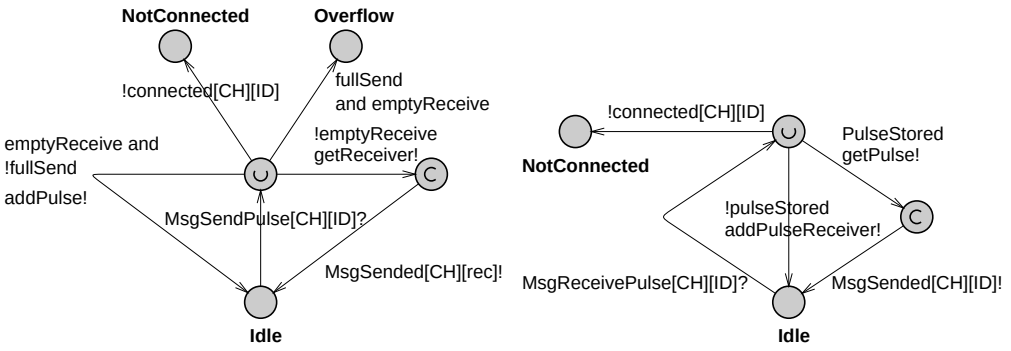


Abbildung 3.46: Modellierung eines Kanals (Senden und Empfangen von Pulsen)

ID benutzt werden kann, muss zuvor eine Verbindung mit der Zuweisung $connected[CH][ID] := true$ hergestellt werden. Versuche, den Kanal vorher zu benutzen, führen in den Fehlerzustand *NotConnected*. Dieser Zustand wird auf Erreichbarkeit geprüft. Es ist nicht nötig, andere Transitionen eines Kanals durch eine entsprechende Bedingung zu sperren, wenn keine Verbindung vorliegt, weil die Erreichbarkeit des Fehlerzustandes davon unberührt bleibt. Beispielsweise brauchen abgehende Transitionen des Zustands A der Abbildung 3.45 links oben nicht mit einer Bedingung versehen werden, die prüft, ob das Flag $connected[CH][ID]$ gesetzt ist. Wird die dynamische Erzeugung von Kanälen und Verbindungen nicht modelliert, entfallen entsprechende Zustände und Transitionen. Initialzustand des Automaten ist dann $Idle$.

Auffällig ist, dass der Fehlerzustand $Overflow$ nur bei Zugriffen auf die Warteschlange *SendBlocked* abgefragt wird. Wie in Abschnitt 3.4.7 ausgeführt wird, kann die Größe der anderen Warteschlangen abgeschätzt werden. Da Clients Pulse versenden können, ohne selbst blockiert zu werden, ist die Größe von *SendBlocked* potentiell unbegrenzt.

Der Zeitautomat verlässt den zentralen Wartezustand, wenn von anderen Threads Kommunikationsaktionen ausgelöst werden. Dieses sind im einzelnen:

- $MsgSend\,[CH]\,[ID]?$ (Abb. 3.45 links oben):
  Enthält die Warteschlange *ReceiveBlocked* keinen Server, wird die ID des sendenden Threads der Warteschlange *SendBlocked* hinzugefügt. Ansonsten findet die Synchronisation von Client und Server über $MsgReceived\,[CH][id]!$ und $MsgSended[CH][rec]!$ statt. Die Variable rec wird durch den Aufruf der Aktion $getReceiver!$ gesetzt. Der Server wird dabei aus *ReceiveBlocked* gelöscht.

- $MsgReceive\,[CH]\,[ID]?$ (Abb. 3.45 rechts oben):
  Enthält *SendBlocked* keine wartenden Clients, wird die ID des Threads *ReceiveBlocked* hinzugefügt. Enthält sie an erster Stelle einen Client mit einer Nachricht, findet die Synchronisation durch die Benachrichtigung von Client ($MsgReceived[CH][id]!$) und Server ($MsgSended[CH]$ $[rec]!$) statt und der Datenaustausch erfolgt.

- $MsgReply\,[CH]\,[ID]?$ (Abb. 3.45 unten):
  Der zugehörige Client wurde im Server gespeichert und wird über die

Variable *Grec* an den Kanal übergeben. Die Blockierung des Clients wird durch ein $MsgReplied[CH]$ $[Grec]$! aufgehoben.

- $MsgSendPulse\,[CH]\,[ID]$? (Abb. 3.46 links):
  Ist kein Server bereit (kein Eintrag in *Receive Blocked*), wird die negierte ID des Threads in *SendBlocked* eingefügt. Ansonsten wird der entsprechende Eintrag des Servers gelöscht und seine Blockierung durch ein $MsgSended[CH][rec]$! aufgehoben. Die Variable *rec* wird wie oben durch einen Aufruf von *getReceiver*! ermittelt. Der Client wird über die Weiterleitung des Pulses nicht wie bei einfachen Nachrichten informiert, da er durch das Versenden nicht blockiert.

- $MsgReceivePulse\,[CH]\,[ID]$? (Abb. 3.46 rechts):
  Befindet sich in *SendBlocked* kein Puls, wird die negierte Thread ID des Servers in *ReceiveBlocked* eingefügt. Damit wird angezeigt, dass nur Pulse empfangen werden können. Ansonsten wird die Blockierung des Threads durch eine Aktion $MsgSended[CH][ID]$! sofort wieder aufgehoben.

3.4.6 Durchführung der Verifikation

Nach der Generierung eines Zeitautomaten-Modells können folgende Eigenschaften verifiziert werden:

- *Deadlines von Tasks*
  Es wird für ein Trigger-Response-Paar überprüft, ob die benutzerdefinierte Deadline eingehalten wird, indem die Erreichbarkeit eines Zustandes geprüft wird, der nur bei einer Überschreitung einer Deadline betreten wird.

- *Kontrolle auf einen Deadlock*
  ($A[]$ *deadlock imply ThreadA.Final and ThreadB.Final and* ...):
  Die einzig zulässige Verklemmung tritt auf, wenn alle Threads abgearbeitet sind und ihren Endzustand erreicht haben (z.B. $ThreadA.Final$), da dann das System zum Stillstand kommt. Alle anderen Deadlocks deuten auf einen Fehler im Systemdesign hin.

- *Erreichbarkeit von Fehlerzuständen*
  ($A[]$ $not(ChannelA.NotConnected$ *or* $ChannelA.Overflow$ *or* ...)):

Fehlerzustände werden bei einem fehlerhaften Verbindungsaufbau oder einem Überlauf einer Warteschlange erreicht.

- *Terminierung der Threads*
  $(A <> (ThreadA.Final\ and\ ThreadB.Final\ and\ \ldots))$: Ist ein Thread mit einem Endzustand versehen, kann dessen zwangsläufiges Erreichen überprüft werden.

3.4.7 Optimierung der Abbildung

Besonders kritisch für die Größe des Zustandsraums sind die Warteschlangen der Kanäle. Der Inhalt einer Warteschlange wird in UPPAAL durch ein Feld von ganzzahligen Variablen verwaltet. Im allgemeinen Fall besitzt ein Kanal drei Warteschlangen (*SendBlocked*, *ReceiveBlocked* und *ReplyBlocked*). Je nach Systemarchitektur können Warteschlangen entfallen oder Automaten vereinfacht werden:

- In der Datenstruktur *ReplyBlocked* werden in QNX blockierte Clients verwaltet, deren Anfrage bereits von einem Server bearbeitet wird. Wird die ID des Clients dem Server übergeben, kann auf diese Datenstruktur verzichtet werden.

- Nimmt ein Kanal nur Pulse entgegen, können in Abbildung 3.45 und 3.46 Hilfszustände entfernt werden, die das Senden einer blockierenden Nachricht verwalten.

- Übermittelt ein Kanal keine Pulse, kann auf entsprechende Hilfszustände verzichtet werden.

- Die Größe der Warteschlange *ReceiveBlocked* entspricht der Anzahl der Server dieses Kanals. Ist nur ein Server vorhanden, wird die Warteschlange durch eine einfache boolesche Variable ersetzt.

- Die Größe der Warteschlange *SendBlocked* ist prinzipiell unbeschränkt groß, wenn im System Pulse verschickt werden: Ein Client kann beliebig viele Pulse verschicken, ohne selbst blockiert zu werden. Dieses führt sowohl im UPPAAL-Modell als auch in realen Systemen dazu, dass diese Warteschlange alle Speichergrenzen sprengt. In UPPAAL wird in diesem Fall ein Zustand $Overflow$ erreicht. Die Größe der Warteschlange

wird per Parameter vorgegeben. Im Fall eines Überlaufens kann versucht werden, die Warteschlange zu vergrößern. Sollte dies kein Erfolg haben, liegt ein Design-Fehler des Programms vor, da die Serverkapazität nicht zur Kapazität der Clients passt.

Im Scheduler-Automaten sind ebenfalls Warteschlangen vorgesehen, wenn es mehr als einen Thread pro Prioritätsstufe gibt. Ist nur maximal ein Thread pro Priorität vorhanden, können die Warteschlangen durch eine einfache Variable ersetzt werden. Prioritätsstufen, die nicht verwendet werden, können entfallen.

3.4.8 Beispiel (Fortsetzung)

Das Modellierungsbeispiel aus Abschnitt 2.3.3 wurde automatisch in ein Automatensystem für UPPAAL umgesetzt. Dabei wurde ein XML-Zwischenformat genutzt, in dem sich QNX-spezifische Modelle effizient abspeichern lassen. Die Umsetzung führt zu einem UPPAAL-System, das in etwa aus 150 Zuständen und 250 Transitionen besteht. Die Verifikation der oben beschriebenen Anfragen liegt für dieses System auf einem handelsüblichen PC im Sekundenbereich.

Generell ist es schwierig, eine Faustformel für den Aufwand der Verifikation von Modellen anzugeben. Die Größe des Zustandsraums, der beim Model-Checking aufgebaut wird, hängt weniger von der Menge der verwendeten Zustände und Transitionen im Modell ab, sondern weitaus mehr von der Anzahl und Größe der verwendeten Variablen und dem Grad des Nichtdeterminismus, den ein Modell aufweist. Enthält ein Modell nur deterministische Abläufe, bleibt der Zustandsraum für einen Model-Checker beherrschbar. Da von einer Steuerung erwartet wird, dass sie deterministisch reagiert, sollte Nicht-Determinismus soweit wie möglich vermieden werden. Allerdings lässt sich die Verwendung von nicht-deterministischen Konstrukten oft nur schwer vermeiden, wenn z. B. Teile der Systemumwelt berücksichtigt werden müssen.

Kapitel 4

Analyse der Schedulability

Im vorherigen Kapitel wurde beschrieben, wie sich Echtzeitsysteme mit Hilfe einer dynamischen Analyse verifizieren lassen. Diese Art der Analyse hat den Nachteil, dass die Modellgröße begrenzt ist, bei der eine Verifikation terminiert. Werden größere Software-Systeme entworfen, ist eine umfassende dynamische Analyse oft nicht durchführbar. In diesem Fall wird zunächst ein größeres Gesamtmodell entworfen und die dynamische Analyse auf Teilmodelle angewendet. Trotzdem ist es nicht nötig, für große Modelle völlig auf eine automatische Analyse zu verzichten.

In diesem Fall lassen sich Verfahren der Schedulability-Analyse anwenden [BW01, Kop97]. In der Literatur gibt es eine große Vielzahl von Algorithmen, die eine solche Analyse durchführen. In dieser Arbeit wurde ein Verfahren herausgegriffen, das weit verbreitet ist. Dieses Verfahren aus der Literatur wird ausführlich im nächsten Abschnitt vorgestellt, um dabei zentrale Begriffe einzuführen, die bei einer Modellierung zu berücksichtigen sind. Im Anschluss daran wird aufgezeigt, wie sich solche Verfahren in eine UML-Modellierung integrieren lassen.

Die Analyse erfolgt dabei auf eine ganz andere Art als in den vorherigen Kapiteln. Die statische Antwortzeitanalyse zielt jedoch wie in den vorausgegangenen Kapiteln darauf ab, die zeitliche Einhaltung von Deadlines zu überprüfen.

4.1 Modellierung von Scheduling-Aspekten

Echtzeitsysteme bestehen im Allgemeinen aus einer Anzahl parallel ablaufender Programme, den Tasks. Diese werden von Prozessen und Threads ausgeführt. Nur durch eine Parallelisierung der Aufgaben lässt sich eine gute Performance erzielen. Tasks laufen allerdings nicht unabhängig voneinander ab, sondern greifen auf gemeinsame Ressourcen zu. Eine wichtige Ressource sind Prozessoren, die Rechenzeit zur Verfügung stellen. Da die korrekte Ausführung eines Tasks auch von seinem Zeitverhalten abhängt, ist ein kontrollierter Zugriff auf Ressourcen notwendig, der als *Scheduling* bezeichnet wird. Die Reihenfolge des Zugriffs von Tasks auf Ressourcen wird in einem Zeitplan, dem *Schedule*, festgehalten. Dieser wird mit Hilfe eines *Scheduling-Algorithmus* aufgestellt. Ein Schedule, der vor der Laufzeit eines Systems ermittelt wird, heißt *statisch*. Wird er erst zur Laufzeit festgelegt, nennt man ihn *dynamisch*.

Die optimale Erstellung eines Schedules ist von vielen Randbedingungen abhängig, die durch ein Anwendungsgebiet und ein konkretes Echtzeitbetriebssystem gegeben sind. In der Literatur lässt sich eine Fülle von Verfahren finden, die sich vor allem aus der Kombinationsvielfalt dieser Randbedingungen erklären lässt. Diese Arbeit beschränkt sich exemplarisch auf die Darstellung eines wichtigen Verfahrens, mit dem viele praxisnahe Anwendungen behandelt werden können.

4.1.1 Scheduling für prozessbasierte Systeme

Es ist durchaus möglich, viele zyklische Echtzeitanwendungen mit *einem* Prozess/Thread zu realisieren, indem dieser in Haupt- und Unterzyklen unterteilt wird. Allerdings ist dieses Verfahren unflexibel und führt bei Tasks mit variablen Ausführungszeiten zu einer schlechten Prozessornutzung. Komplexe Echtzeitanwendungen werden daher oft als prozessbasiertes System realisiert. Dazu wird ein Scheduler benötigt, der Prozesse koordiniert und ihnen Rechenzeit zuweist. Der Scheduler ist in der Regel Teil des Betriebssystems.

Ein Prozess τ_i wird durch ein bestimmtes Ereignis, wie z. B. einem periodischen Zeitsignal mit der Periode T_i aktiviert und wartet dann auf die Zuteilung von Rechenzeit auf einem Prozessor. Diese Rechenzeit muss mindestens einer Worst-Case-Execution-Time (WCET) C_i entsprechen, die zur Ausführung benötigt wird und sollte vor Ablauf einer Deadline D_i vollständig bereitgestellt werden.

Um die Einhaltung von Deadlines zu gewährleisten, wird eine Priorität P_i für jeden Prozess τ_i vergeben. Der Scheduler übergibt die Kontrolle an denjenigen Prozess, der die höchste Priorität hat und nicht blockiert ist, z. B. weil er auf eine andere Ressource wartet. Dabei gibt es Implementierungen, in denen der Scheduler erst aktiv wird, wenn der aktuelle Prozess terminiert oder blockiert ist. Im Folgenden wird davon ausgegangen, dass der Scheduler *sofort* einschreitet, wenn ein Prozess mit höherer Priorität ausführbar wird. In diesem Fall wird der aktuelle Prozess möglichst ohne Zeitverzögerung unterbrochen (*Preemption*).

Aufgabe eines Scheduling-Algorithmus ist die Vergabe von Prioritäten, so dass alle Deadlines garantiert werden. Das heißt während T_i, C_i und D_i anwendungsspezifische Parameter der Prozesse sind, wird P_i vom Scheduling-Algorithmus festgelegt. Es wird dabei zwischen Systemen unterschieden, in denen sich die Prioritäten während der Laufzeit nicht ändern (*Fixed Priority Scheduling*) und solchen, bei denen Prioritäten dynamisch festgelegt werden. Die Scheduling-Algorithmen werden dabei nach den Strategien benannt, die bei der Festlegung der Prioritäten angewendet werden. Da ein Schedule unter Berücksichtigung von Prioritäten erst zur Laufzeit erstellt wird, handelt es sich sowohl bei festen als auch bei dynamischen Prioritäten um ein *dynamisches* Scheduling.

Zusammenzufassend wird nun ein grundlegendes Prozessmodell definiert. Dieses Grundmodell wird schrittweise zu einem realitätsnahen Prozessmodell erweitert [BW01].

Prozessmodell. Wir betrachten zunächst prozess- und prioritätsorientierte Systeme mit einer festen Anzahl von N Prozessen $\tau_1, \ldots, \tau_N$ und einem Prozessor. Die Prozesse werden durch folgende Größen charakterisiert:

T_i : Minimale Zeit zwischen zwei Aktivierungen von τ_i (bei periodischen Prozessen ist dies die Periode)

D_i : Deadline von τ_i

C_i : Maximale Ausführungszeit von τ_i, wenn der Prozessor durchgängig exklusiv genutzt werden kann

P_i : Priorität von τ_i

Es wird vorläufig von rein periodischen Prozessen mit strengen Deadlines ausgegangen, von denen jeweils maximal eine Instanz aktiv ist, d. h. es gilt $D_i \leq T_i$. Eine Blockierung durch gemeinsam genutzte Ressourcen wird noch nicht betrachtet und zum Zeitpunkt des Systemstarts werden alle Prozesse gleichzeitig aktiviert.

Vergabe von Prioritäten. Prinzipiell kann die Verteilung der Prioritäten auf willkürliche Weise mit unterschiedlicher Auswirkung auf das Scheduling durchgeführt werden. Folgende Strategien sind in der Praxis besonders häufig anzutreffen:

Rate Monotonic Priority Order (RMPO). Dem Prozess mit der kürzesten Periode wird die höchste Priorität zugewiesen:

$$T_i < T_j \Rightarrow P_i > P_j.$$

.

Deadline Monotonic Priority Order (DMPO). Dem Prozess mit der kürzesten Deadline wird die höchste Priorität zugewiesen:

$$D_i < D_j \Rightarrow P_i > P_j.$$

.

Earliest Deadline First (EDF). Der Prozess, dessen Deadline zur Laufzeit zeitlich am nächsten liegt, erhält die höchste Priorität.

Die ersten beiden Verfahren sind statisch, da sich die Prioritäten zur Laufzeit nicht ändern. Das dritte Verfahren ist dynamisch und lässt in der Theorie eine optimale Ausnutzung des Prozessors zu. Allerdings wird dieser theoretische Vorteil durch eine aufwendige Realisierung des Schedulers erkauft, der zur Laufzeit selbst Rechenzeit benötigt. Daher sind in der Praxis oft die ersten beiden Prioritätenordnungen effizienter.

4.1.2 Analyse der Schedulability

Nachdem ein einfaches Prozessmodell aufgestellt wurde, wird nun bestimmt, ob eine bestimmte Prioritätsordnung das Einhalten aller Deadlines garantiert. Dazu wird folgende Definition benötigt:

Schedulability. Für eine Menge von Tasks ist die Schedulability gegeben, wenn ein Scheduling-Algorithmus existiert, bei dessen Anwendung alle Tasks stets ihre Deadline einhalten.

Bei der Analyse ist zu beachten, dass das Ergebnis immer auch vom Prozessmodell abhängig ist. Fließen andere Annahmen in das Modell ein, muss die Analyse abgeändert werden. Insbesondere nehmen Analyseverfahren pessimistische Abschätzungen vor. Wird ein Modell verfeinert, kann es dazu führen, dass ein System, für das vorher keine Schedulability gegeben war, doch sicher ausführbar wird.

Es gibt verschiedene Verfahren für die Überprüfung, ob eine Prioritätenordnung die Einhaltung aller Deadlines garantiert. Eine grundlegende Arbeit dazu ist [LL73]. Ein allgemein verbreiteter Ansatz berechnet die maximale Antwortzeit jedes Tasks nach dessen Aktivierung. Ein Vergleich zwischen den maximalen Antwortzeiten und den Deadlines ergibt, ob das System zeitlich korrekt abläuft. Die Herleitung der Formeln ist [JP86] entnommen.

Ausgangspunkt der Berechnung sind die nachfolgenden Werte:

R_i : Maximale Antwortzeit von Prozess τ_i nach dessen Aktivierung.

I_i : Maximale Unterbrechungszeit (Interferenz) durch Prozesse höherer Priorität innerhalb der Antwortzeit R_i.

Es gilt dann

$$R_i = C_i + I_i.$$

Es wird nun von den beiden statischen Prioritätenordnungen RMPO und DMPO ausgegangen. Gesucht wird der Zeitpunkt, zu dem die Antwortzeit eines Prozesses am längsten ist. Dieser so genannte *Critical Instant* ergibt sich, wenn *alle* Prozesse rechenwillig sind. In diesem Fall wird ein Prozess τ_j mit einer höheren Priorität ($P_j > P_i$) innerhalb der Antwortzeit R_i maximal $\left\lceil \frac{R_i}{T_j} \right\rceil$-mal aktiviert. Der Prozess τ_i wird dadurch für eine Dauer von $C_j \left\lceil \frac{R_i}{T_j} \right\rceil$ Zeiteinheiten unterbrochen. Die gesamte Interferenz resultiert daher aus der Summe dieser Einzelinterferenzen und wird in die obige Formel eingesetzt:

$$R_i = C_i + \sum_{\{j|P_j>P_i\}} C_j \left\lceil \frac{R_i}{T_j} \right\rceil.$$

Diese Formel ist rekursiv. In [Aud93] wird nachgewiesen, dass sich die Lösung als Fixpunkt folgender iterativen Formel ergibt. Ein geeigneter Startwert ist $\omega_i^0 = C_i$.

$$\omega_i^{n+1} = C_i + \sum_{\{j|P_j>P_i\}} C_j \left\lceil \frac{\omega_i^n}{T_j} \right\rceil .$$

Bisher sind Ressourcen im Systemmodell nicht berücksichtigt worden. Das Modell wird nun um eine Menge von K exklusiven Ressourcen erweitert, d. h. eine Ressource kann maximal von einem Prozess benutzt werden. Alle Prozesse, die auf eine belegte Ressource zugreifen wollen, sind blockiert. Die Antwortzeit verlängert sich damit um die Blockierzeit B_i:

$$\omega_i^{n+1} = C_i + B_i + \sum_{\{j|P_j>P_i\}} C_j \left\lceil \frac{\omega_i^n}{T_j} \right\rceil .$$

Dabei gehen in B_i nur Blockierzeiten ein, die durch einen Prozess τ_j mit einer niedrigeren Priorität als der von τ_i ($P_j < P_i$) verursacht werden. Unterbrechungen durch Prozesse mit höherer Priorität wurden bereits in der Interferenz berücksichtigt. Dabei ist es unerheblich, ob diese höher priorisierten Prozesse zur Unterbrechungszeit auch Ressourcen belegen oder nicht.

Die Berechnung von B_i hängt vom Protokoll ab, mit dem der Zugriff auf Ressourcen geregelt wird. Generell ist es wünschenswert, B_i mit Hilfe eines Protokolls so kurz wie möglich zu halten, weil es in dieser Zeit zu einem Bruch im Prioritätenschema kommt, d. h. Prozesse höherer Priorität werden von Prozessen mit niedriger Priorität unterbrochen. Insbesondere sollte es vermieden werden, dass der niedrig priorisierte Prozess seinerseits vor der Freigabe der Ressource durch weitere Prozesse, die nicht an Ressourcenzugriffen beteiligt sind, unterbrochen wird. In diesem Fall könnte der hoch priorisierte Prozess dauerhaft blockiert bleiben. Ein weit verbreitetes Protokoll zur Vermeidung dieser Effekte ist das Folgende.

Priority Ceiling Protocol. Das Priority Ceiling Protocol [SRL90] ist eine Weiterentwicklung des *Priority Inheritance Protocols*. Im letztgenannten Protokoll wird ein blockierender Prozess für die Dauer der Blockierung auf die Priorität des blockierten Prozesses angehoben. Allerdings kommt es so zu häufigen Prioritätswechseln und vor allem sind Deadlocks nicht ausgeschlossen [BW01].

Für das Priority Ceiling Protocol gelten drei Regeln:

- Jeder Prozess τ_i hat eine feste Basispriorität P_i.

- Jede Ressource hat eine feste Ceiling-Priorität, die sich aus dem Maximum der Prioritäten aller Prozesse ergibt, die sie benutzen können.
- Jeder Prozess hat eine dynamische Priorität. Diese ist jeweils das Maximum aus seiner Basispriorität und den Ceiling-Prioritäten, die er gerade verwendet.

In [SRL90] wird nachgewiesen, dass die Implementierung dieses Protokolls die folgenden Eigenschaften sichert:

- Priority Inversion wird verhindert: Es kann nicht vorkommen, dass ein hoch priorisierter Prozess auf eine Ressource wartet, die nicht freigegeben wird, weil der blockierende Prozess selbst durch einen Prozess mit mittlerer Priorität unterbrochen wird.
- Deadlocks und transitive Folgen von Blockierungen werden verhindert.
- Ein Prozess kann während seiner Ausführung höchstens einmal von einem Prozess mit niedriger Priorität unterbrochen werden.
- Der exklusive Zugriff auf Ressourcen ist sichergestellt. Semaphoren sind nicht nötig.

Eine Blockierung findet nur statt, solange ein niedrig priorisierter Prozess aufgrund einer Ressourcennutzung mit seiner dynamischen Priorität einen hoch priorisierten Prozess unterbricht. Die Zeit, in der eine Ressource ohne die Möglichkeit eines Abbruchs reserviert ist, wird als *kritischer Bereich* bezeichnet. $Crit_{j,s}$ definiert den kritischen Bereich einer Ressource s, die von einem Prozess τ_j benutzt wird. Damit ergibt sich B_i als längster kritischer Bereich, der eine Priorität höher als P_i verleihen kann:

$$B_i = \max_{j,k \in \{1,\ldots,N\}, s \in \{1,\ldots,K\}} \{Crit_{j,s} | P_j < P_i \wedge P_k \geq P_i \wedge \tau_j, \tau_k \in uses\,(s)\}\,.$$

Bisher galt $D_i \leq T_i$. Wird diese Forderung fallen gelassen, resultieren sich überlappende Aktivierungen von Prozessen. Es wird dabei vorausgesetzt, dass diese überlappenden Aktivierungen in der Reihenfolge ihres Auftretens abgearbeitet werden. Unterschiedliche Prozessaktivierungen können verschiedene Antwortzeiten haben. Zur Berechnung der maximalen Antwortzeit wird

nach [TBW94] ein Zeitfenster $\omega(q)$ definiert, das q zusätzliche Überlappungen eines Prozesses enthält. Diese insgesamt $q+1$ Aktivierungen benötigen ohne Blockierung oder Interferenz $(q+1)C_i$ Zeiteinheiten. Ansonsten kann die Länge des Zeitfensters $\omega(q)$ analog zu oben iterativ ermittelt werden:

$$\omega_i^{n+1}(q) = (q+1)C_i + B_i + \sum_{\{j|P_j>P_i\}} C_j \left\lceil \frac{\omega_i^n(q)}{T_j} \right\rceil .$$

Die Antwortzeit einer bestimmten (der q-ten) Aktivierung erhält man, indem die Perioden der vorherigen Aktivierungen subtrahiert werden:

$$R_i(q) = \omega_i(q) - qT_i.$$

Es müssen die Antwortzeiten $R_i(q)$ solange berechnet werden, bis für ein Q_i gilt: $R_i(q) \leq T_i$. In diesem Fall liegt für die nächste Aktivierung keine Überlappung mehr vor. Ebenfalls abgebrochen wird, wenn $R_i(q) > D_i$, da in diesem Fall die Deadline verletzt ist. Damit ergibt sich die maximale Antwortzeit als

$$R_i = \max_{q\in\{0,\ldots,Q_i\}} R_i(q) .$$

Als letzte Erweiterung sollen Verzögerungen bei der Auslösung periodischer Prozesse berücksichtigt werden. In realen Systemen ist eine absolut präzise Auslösung von Prozessen nicht zu gewährleisten. Mit einem *Jitter* J_i wird die maximale Verzögerung vom rechnerischen Zeitpunkt der Aktivierung bezeichnet. Eine Verzögerung bei der Prozessauslösung bewirkt eine Verlängerung der Zeit, in der ein Prozess unterbrochen werden kann. Dieses hat folgende Auswirkung auf die Formel zur Berechnung der Antwortzeit [TBW94]:

$$\omega_i^{n+1}(q) = (q+1)C_i + B_i + \sum_{\{j|P_j>P_i\}} C_j \left\lceil \frac{\omega_i^n(q) + J_i}{T_j} \right\rceil .$$

Es wurde damit ein gängiges Verfahren zur Überprüfung der Schedulability unter ganz bestimmten Voraussetzungen vorgestellt. Mit diesem Verfahren werden viele praxisnahe Beispiele abgedeckt. Allerdings ist damit die Theorie der Schedulability-Analyse nicht ansatzweise erschöpfend behandelt. Im Folgenden wird exemplarisch gezeigt, wie sich diese Algorithmen in den Kontext einer UML-basierten Modellierung einfügen lassen. Dabei müssen vor allem

die beschriebenen Parameter in ein UML-Modell integriert werden. Eine Erweiterung um andere Algorithmen der Literatur ist problemlos möglich.

4.2 Modellierung mit UML

Eine Analyse der Schedulability erfolgt durch Anwendung des oben vorgestellten oder ähnlicher Algorithmen. Das wirft die Frage auf, ob sich derartige Algorithmen auch auf eine Modellierung mit UML anwenden lassen. Dies wäre wünschenswert, da man so schnell automatisiert ermitteln könnte, welche Folgen eine Änderung im Design auf die Schedulability haben würde. Natürlich müssen die Modelle eine bestimmte Form haben, um analysierbar zu sein. Insbesondere müssen alle benötigten Eingabeparameter und deren Beziehungen enthalten sein.

Das führt zur grundlegenden Fragestellung, ob UML eine geeignete Sprache zur Darstellung von Scheduling-Aspekten ist. Betrachtet man nur den Kern der UML wird eine Modellierung von diesen Leistungsmerkmalen nur schwach unterstützt. Mit Hilfe der Erweiterungsmechanismen von UML lassen sich allerdings ohne weiteres Konstrukte einführen, welche diese Lücke schließen. Damit dieses in standardisierter Form passiert, wurde eine entsprechende Erweiterung von der OMG veröffentlicht.

Das UML-Profil für Schedulability, Performance and Time [Obj05]. Um die UML zur Modellierung von Echtzeitsystemen anwenden zu können, wurde 2001 das UML-Profil für *Schedulability, Performance and Time* [Obj05] (im folgenden: *SPT-Profil*) von der OMG veröffentlicht, das standardisierte Modellelemente für die Spezifikation von Echtzeitsystemen zur Verfügung stellt. Das UML-Profil soll die Konstruktion von Modellen unterstützen, die quantitative Aussagen über Schedulability, Zeit und Performance enthalten, und den standardisierten Austausch dieser Modelle zwischen Modellierungs- und Analysewerkzeugen ermöglichen.

Das UML-Profil verwendet die vorhandenen Erweiterungsmechanismen der UML, nämlich Stereotypen und Tagged Values, um in statische und dynamische Modelle Scheduling- und Performance-Informationen zu integrieren. Dabei kann ein Stereotyp auf verschiedene Modellelemente angewandt werden, so dass es viele unterschiedliche Möglichkeiten gibt, die analyserelevanten Informationen in das Modell einzubringen.

Den Kern des Profils bildet das *General Resource Model (GRM)*, in dem grundlegende Konzepte zur Modellierung von Ressourcen, Nebenläufigkeit und Zeit eingeführt werden. Basierend auf dem GRM werden Sub-Profile für die Schedulability- ($SAprofile$) und Performance-Analyse ($PAprofile$) definiert.

Grenzen des UML-Profils. Wenn ein standardisiertes UML-Profil für Echtzeitaspekte existiert, was bleibt dann noch zu tun? Bisher fehlte die Umsetzung in einem konkreten Analyseverfahren. Das Profil liefert eine Sammlung von Begriffen und zugehörigen Markierungen in Form von Stereotypen. Allerdings benötigt kein Analyseverfahren alle Stereotypen und eine Analyse eines Modells, in dem diese Stereotypen willkürlich eingestreut werden, ist nicht möglich. Das UML-Profil liefert *keine* Vorschrift, wie Modelle konstruiert werden müssen, die nach einem bestimmten Algorithmus analysiert werden sollen. Hinweise zur Anwendung für reale Analyseprobleme werden nicht gegeben. Tatsächlich dürfte mit [DHK03] die erste praktische Umsetzung und Implementierung des Profils vorliegen.

Hinweis. Für das SPT-Profil gibt es derzeit kein deutschsprachiges Referenzwerk, das als Maßstab für die Übersetzung neu eingeführter Begriffe dienen könnte. Da es die wesentliche Zielsetzung des Profils ist, neue Modellierungskonstrukte in Form von Stereotypen bereitzustellen, werden in dieser Arbeit die englischen Begriffe beibehalten. Es wird in Kauf genommen, dass dieser Text englische und deutsche Ausdrücke mischt. Eine Übersetzung würde den Bezug zum Standard unnötig erschweren.

4.2.1 Modelle und Metamodelle

Während einer Analyse der Schedulability spielen drei Sichtweisen eine Rolle, deren Aufbau in Metamodellen beschrieben wird. Es wird hier ein kurzer Überblick über die Zusammenhänge gegeben.

Zunächst kann der Modellierer beliebige *konkrete UML-Modelle* erstellen, die dem *UML-Metamodell* entsprechen, so wie es im OMG-Standard definiert ist. Dazu ist im Regelfall keine explizite Kenntnis des Metamodells nötig. Es ist Aufgabe eines Modellierungstools, nur solche Modelle zuzulassen, die konsistent zum UML-Metamodell sind.

Derartige Modelle sind zunächst zu ausdrucksschwach, um eine Analyse der Schedulability zu ermöglichen. Es werden zusätzliche Konstrukte benötigt. Der Zusammenhang zwischen diesen anwendungsspezifischen Konstrukten wird in einem *Domain-Metamodell* beschrieben. Es kann herangezogen werden, wenn es gilt, in einem konkretem UML-Modell die Verknüpfungen zwischen Scheduling-Stereotypen zu rekonstruieren: Findet sich in einem konkreten Modell eine Verbindung zwischen zwei Stereotypen, lässt sich dem Domain-Metamodell entnehmen, ob diese Verbindung auch zulässig ist. Die Definition dieses *Domain-Metamodells* ist Teil dieser Arbeit und führt zusammen mit einer genauen Beschreibung seiner Komponenten zu einem neuen UML-Profil (*ExecutableSAprofile*). Dabei wird das SPT-Profil als Stereotypen-Katalog verwendet, aus dem die benötigten Konstrukte ausgewählt werden. Proprietäre Erweiterungen werden äußerst sparsam eingesetzt, obwohl das SPT-Profil selbst keinen Anspruch auf Vollständigkeit erhebt. Der Zusammenhang zwischen diesem Profil und dem SPT-Profil wird unten beschrieben.

Das Domain-Metamodell ist die Grundlage für ein *Analyse-Metamodell*. Beide Modelle sind stark verwandt. Im Analyse-Metamodell werden gegenüber dem Domain-Metamodell Änderungen vorgenommen, die eine Analyse vereinfachen. Beispielsweise können zusätzliche Assoziationen erlaubt werden, welche die Navigation erleichtern. Im Domain-Modell sind konzeptionelle Klassen vorgesehen, denen kein Stereotyp zugeordnet ist. Diese können aus pragmatischen Gründen im Analysemodell aufgrund direkter Verbindungen zwischen Stereotypen entfallen. Das Analysemodell entspricht der Objektstruktur, die in einer Implementierung während der Analyse aufgebaut wird. Das Analyse-Metamodell spiegelt sich in der Klassenstruktur des realisierten Programmcodes wieder und stellt eine Vereinfachung des Domain-Metamodells da. Ausschlaggebend sind Implementierungsaspekte, auf die im Folgenden nicht weiter eingegangen wird. Im Text werden Domain-Modell und Analysemodell in der Regel synonym verwendet.

4.2.2 Modellierung von Scheduling-Aspekten

Im Folgenden werden die Konstrukte beschrieben, die bei der Modellierung von Echtzeitsystemen zur Schedulability-Analyse angewendet werden können. Es handelt sich um eine anwendungsspezifische Auswahl aus dem SPT-Profil [Obj05], d.h. es finden sich dort fast alle hier verwendeten Elemente und Zusammenhänge und noch viele weitere wieder, die für diese An-

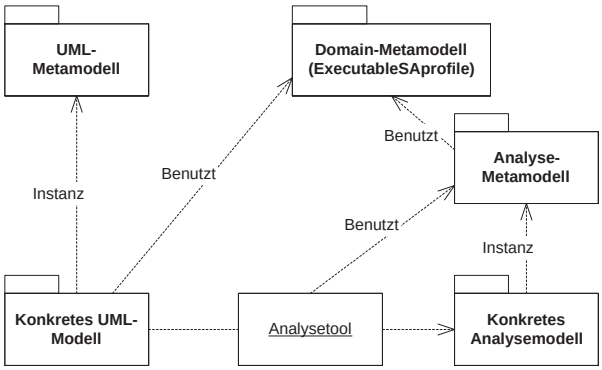


Abbildung 4.1: Beziehungen zwischen Metamodellen und Modellinstanzen

Abbildung 4.2: Importbeziehungen des ExecutableSAprofils.

wendung irrelevant sind. Insbesondere wird dabei auf Elemente der Pakete $RT\,resourceModeling$ (Ressourcenmodellierung), $RT\,timeModeling$ (Zeitmodellierung) und $SAprofile$ (Profil für Schedulability-Analyse) zurückgegriffen (Abb. 4.2). Wir definieren damit ein neues Profil, das sich durch einen deutlich *reduzierten* Umfang gegenüber [Obj05] auszeichnet - und der Eigenschaft, dass es *ausführbar* im Sinne von analysierbar ist. Daher wird es als *ExecutableSAprofile* (*ESA*) bezeichnet.

Die folgende Tabelle gibt eine Übersicht über die Stereotypen des ESA-Profils. Den Stereotypen sind Eigenschaftswerte zugeordnet, die für eine Analyse benötigt werden. Weiterhin werden Basisklassen des UML-Metamodells angegeben, auf denen sich ein Stereotyp anwenden lässt. Beispielsweise kann das Stereotyp $SATrigger$ eine Nachricht kennzeichnen. Zusätzlich wird mit Hilfe von $SAOccurence$ ein Auftrittsmuster definiert („Tritt alle $125\mu s$ auf“).

Stereotypen kennzeichnen Bestandteile eines Anwendungsbereiches, d.h. eine Domain-Klasse. Es handelt sich in diesem Zusammenhang um Konzep-

te der Schedulability-Analyse. Ähnlich dem UML-Metamodell werden Beziehungen zwischen Anwendungskonzepten ebenfalls in einem Metamodell dargestellt. In den Abbildungen 4.3 und 4.4 ist der Kern des Metamodells des ESA-Profils abgebildet, der sich aus einer Filterung des Metamodells aus [Obj05] ergibt. Zu beachten ist, dass nicht für jedes Konzept des Anwendungsbereichs ein Stereotyp existiert. Beispielsweise existiert für einen Task das Konzept des *Scheduling Jobs*. Es gibt dafür kein korrespondierendes Stereotyp, da Tasks im UML-Modell über ihren Trigger und die induzierte Response definiert werden.

Die Basisklasse eines Stereotyps ist nicht eindeutig festgelegt. Dieses gibt dem Software-Designer eine große Freiheit beim Modellieren. Ob sich ein bestimmtes Diagramm zur Veranschaulichung der Scheduling-Aspekte einer konkreten Software eignet, bleibt seine Entscheidung. Bei der späteren Analyse werden die Stereotypen aus dem Modell extrahiert und deren Verknüpfungen dem Modell entnommen. Auf dieser Ebene spielen die Basisklassen keine Rolle mehr.

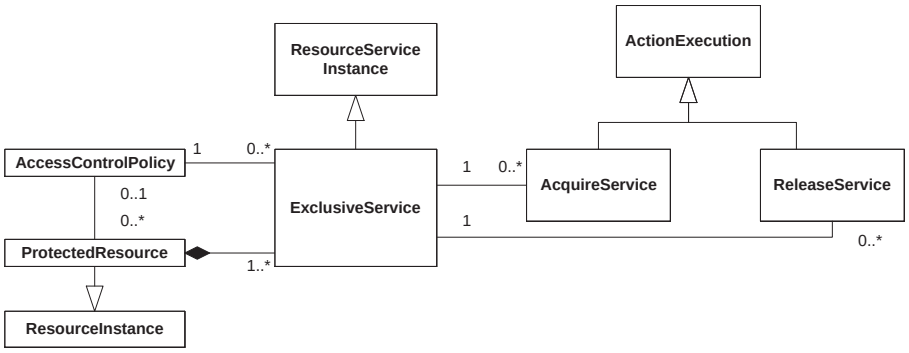


Abbildung 4.3: Ressourcenzugriff (aus [Obj05], 3-10).

Stereotyp	Eigenschaftswerte	Basisklasse	Domain-Klasse
SAEngine	SASchedulingPolicy	Klasse, Objekt Knoten Knoteninstanz	Execution Engine
SASchedulable (ab Version 1.1: SAschedRes)		Instanz/Rolle (aktive) Klasse	Schedulable Resource
SATrigger	SAOccurence	Nachricht Ereignis, Transition	Trigger
SAResponse	RTduration SAAbsDeadline SAPriority *SAJitter*	Methode Nachricht Aktion Action Execution	Response
SAAction	RTduration	Methode Nachricht Aktion Action Execution	SAction
SAResource	SAAcquisation SADeacquisation	Instanz/Rolle Objekt/Klasse Objektzustand	SResource
GRMacquire		wie SAAction	Acquire Service
GRMrelease		wie SAAction	Release Service

Die verwendeten Stereotypen werden in den folgenden Abschnitten ausführlich erläutert.

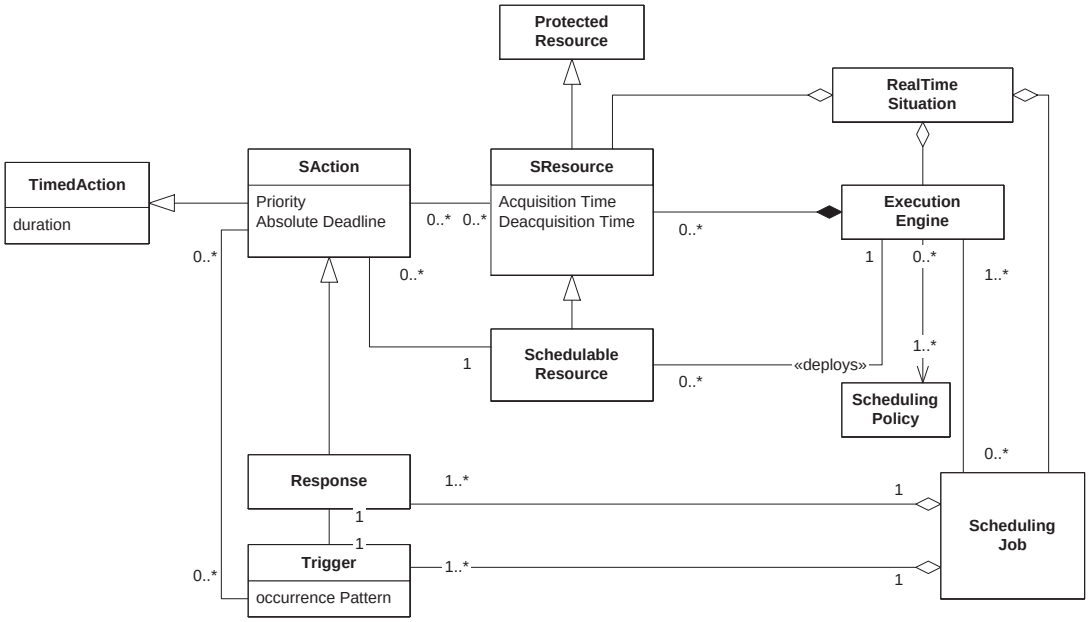


Abbildung 4.4: Schedulability-Modell (aus [Obj05], Abb. 6-1).

4.2.3 Prozesse und Prozessoren

Ein Prozess entspricht einer *Schedulable Resource* (Abb. 4.4). Zur Kennzeichnung von Prozessen wird der Stereotyp $<< SA\,Schedulable >>$ verwendet. Es können damit folgende Modellelemente gekennzeichnet werden:

- Eine Instanz oder Rolle in einem Sequenz- oder Kollaborationsdiagramm.
- Ein Objekt oder eine (aktive) Klasse.

Jeder Prozess benötigt einen Prozessor zur Ausführung (*Execution Engine*). Dafür ist das Stereotyp $<< SA\,Engine >>$ des Profils vorgesehen, das auf folgende Elemente angewandt werden kann:

- Ein Knoten oder eine Knoteninstanz in einem Verteilungsdiagramm. Die Zuordnung zwischen Prozess und Prozessor ergibt sich damit intuitiv aus dem Diagramm, da die Prozesse im ausführenden Prozessor-Knoten dargestellt sind.

Abbildung 4.5: Beziehung zwischen Prozessor und Prozess

- Ein Objekt oder eine Klasse. In diesem Fall ist eine explizite Zuordnung der Prozesse zum Prozessor notwendig. Dies geschieht mit Hilfe einer Abhängigkeit, die mit dem Stereotyp $<< GRM\ deploys >>$ gekennzeichnet ist (Abb. 4.5).

Während einer Analyse müssen Prozesse und Prozessoren einander zugeordnet werden - entweder implizit über die graphische Darstellung (Prozessoren umfassen ihre Prozesse) oder über die explizite Abhängigkeit $<< GRM\ deploys >>$.

Das Stereotyp $<< SA\,Engine >>$ für Prozessoren lässt folgende Eigenschaftswerte zu, die für eine Analyse erforderlich sind: Mit $SA\,Scheduling\ Policy$ wird eine Strategie zur Prioritätenvergabe festgelegt. Gültige Werte sind $RateMonotonic$, $DeadlineMonotonic$, $EarliestDeadlineFirst$ und $FixedPriority$. In letzterem Fall muss einem Task explizit eine Priorität zugeordnet werden (siehe unten) .

4.2.4 Trigger und Response

Prozesse führen Tasks (Scheduling Jobs) aus. Im SPT-Profil gibt es allerdings keinen eigenen Stereotyp für die Modellierung eines Tasks. Stattdessen werden sein *Trigger* und die *Response* darauf modelliert. Ein Trigger ist der Auslöser eines Tasks und wird mit $<< SA Trigger >>$ gekennzeichnet. Dieses Stereotyp wird sinnvollerweise auf folgende UML-Elemente angewendet:

- Eine Nachricht in einem Sequenz- oder Kollaborationsdiagramm.
- Ein Ereignis oder eine Transition in einem Zustands- oder Aktivitätsdiagramm.

Es bietet sich an, dass im Falle einer Transition als Trigger eine Default-Transition auf oberster Ebene gewählt wird. Ob es im Einzelfall Sinn macht, andere Transitionen zu kennzeichnen, liegt aber letztlich im Ermessen des Modellierers. Wie weiter unten gefordert wird, sind lediglich unendliche Abläufe durch Zyklen auszuschließen.

Eigenschaftswerte. Für die Bestimmung von Antwortzeiten ist neben den Ausführungszeiten der Tasks das Auftrittsmuster entscheidend. Dieses wird als Wert von *SAOccurrence* angegeben. In [Obj05] wird dafür der Datentyp *RTarrivalPattern* definiert. Hier können periodische Tasks definiert werden (Datenelement *periodic* mit Periodenlänge) und Tasks, für deren Aktivierungsintervalle minimale und maximale Zeitdauern angegeben werden (Datenelement *bounded*). Andere Auftrittsmuster sind vorgesehen, werden aber nicht in diesem Kontext verwendet.

Die Behandlung eines getriggerten Tasks erfolgt durch eine Response, also einem UML-Element, das mit dem Stereotyp << *SAResponse* >> gekennzeichnet ist. Mögliche Kandidaten, um eine Response zu modellieren sind folgende Elemente:

- Eine Methode. In diesem Fall muss der Auslöser des Methodenaufrufs (Nachricht, Ereignis, Transition) als Trigger gekennzeichnet werden. Die Klasse, zu der die Methode gehört, muss mit dem Stereotyp << *SASchedulable* >> als Prozess markiert sein. Die Zuordnung vom Trigger zur getriggerten Methode erfolgt über die Beschriftung der Nachricht bzw. Transition. Dort wird ein entsprechender Methodenaufruf erwartet. Eine andere Möglichkeit ist, dass eine Methode direkt mit einem Aktivitätsdiagramm verknüpft ist. In diesem Fall sollte die übergeordnete Methode als Response gekennzeichnet sein. Der zugehörige Trigger befindet sich dann in der Regel an der Starttransition des Diagramms.

- Eine Nachricht in einem Sequenz- oder Kollaborationsdiagramm. In diesem Fall stimmen Trigger und Response überein. Die eigentliche Antwort ist der durch die Nachricht ausgelöste Ablauf. Die Instanz oder Rolle, die eine Nachricht empfängt, muss mit << *SASchedulable* >> gekennzeichnet sein.

- Eine Aktion in einem Aktivitätsdiagramm. Dann wird der gesamte folgende Ablauf als Response interpretiert. Ihr auslösendes Ereignis muss mit dem Stereotyp eines Triggers versehen sein. Das Aktivitätsdiagramm muss einer Instanz bzw. Klasse untergeordnet sein, die eine Schedulable Resource ist.

- Eine Action Execution. Eine Action Execution hat keine direkte Entsprechung in einem UML-Diagramm. Allerdings kann ein Ausführungs-

balken in einem Sequenzdiagramm als Visualisierung einer Action Execution interpretiert und als Response angesehen werden. In diesem Fall muss die Nachricht, die eine Action Execution auslöst, als Trigger gekennzeichnet sein. Die zugehörige Instanz oder Rolle sollte eine Schedulable Resource sein.

Modellverknüpfungen. Trigger und Response sind einander zugeordnet und definieren einen Task. Dieser wiederum ist mit einem Prozess verknüpft, also einem Element, das mit $<< SA\,Schedulable >>$ gekennzeichnet ist. Die folgende Tabelle fasst die Verknüpfungsmöglichkeiten zusammen.

Trigger	Response	Prozess
Nachricht/Ereignis/Transition	Methode	Klasse
Nachricht	Nachricht	Zielinstanz/-rolle
Ereignis	Transition	Instanz/Klasse
Nachricht	Action Execution	Instanz/Rolle

Auch hier gilt, dass es dem Modellierer überlassen bleibt, zu entscheiden, welche Art der Modellierung seine Intention am besten visualisiert.

Eigenschaftswerte. Die maximale Ausführungszeit C_i eines Tasks kann mit dem Eigenschaftswert *RT duration* der Response definiert werden. Diese entfällt allerdings dann, wenn die Response durch eine Abfolge von Einzelschritten verfeinert dargestellt wird. In diesem Fall wird ein Wert für C_i aus der Modellierung der Einzelaktionen abgeleitet, so wie es weiter unten erläutert wird.

Mit dem Eigenschaftswert *SA AbsDeadline* wird die Deadline D_i eines Tasks spezifiziert. Wenn diese Angabe fehlt, wird angenommen, dass $D_i = T_i$ ist, d.h. die Deadline wird mit der Auftrittsrate des Triggers gleichgesetzt. Dies ist allerdings nur möglich, wenn ein periodischer Task spezifiziert wird.

Im Regelfall werden Prioritäten schematisch nach praxisbewährten Verfahren wie Rate Monotonic oder Deadline Monotonic vergeben. Im Einzelfall kann es sinnvoll sein, eine Priorität für einen Task explizit zu setzen. Dieses erfolgt über den Eigenschaftswert *SA Priority*.

4.2.5 Aktionen

Mit den bisherigen Mitteln lassen sich Tasks durch Trigger und Responses definieren. Priorität, Dauer, Aktivierungsmuster/-rate und Deadline lassen

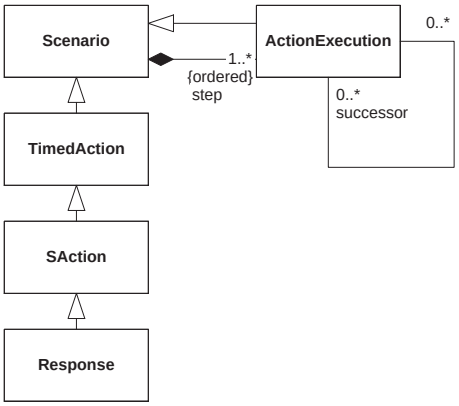


Abbildung 4.6: Verfeinerung von Scheduling-Aktionen

sich angeben. Damit können die wichtigsten Angaben für eine Analyse der Schedulability bereits gemacht werden. Allerdings sind die Modellierungsmöglichkeiten noch nicht besonders flexibel und berücksichtigen keine Verhaltensspezifikationen, die in Zwischenschritte verfeinert sind.

Für einzelne Scheduling-Aktionen sieht das SPT-Profil den Begriff der *SAction* vor. In Abbildung 4.6 sind die Beziehungen der relevanten Konstrukte zusammengefasst. Eine Response führt zur Laufzeit zu einer Serie von Action Executions (Komposition im Modell). Umgekehrt erbt eine Action Execution selbst die Komposition vom Scenario. Damit kann auch sie ihrerseits durch Action Executions verfeinert werden, die über eine Nachfolger-Relation verbunden sind. Auf Modellebene können Action Execution als SAction dargestellt werden. So lässt sich ein *ActionExecution*-Szenario einer Response mit einem Aktivitätsdiagramm modellieren, in dem alle Aktionen als SAction gekennzeichnet sind.

Eine SAction ist die Spezialisierung einer einfachen Aktion. Hinzugefügt wurden Eigenschaftswerte, die auf ihr Scheduling Einfluss nehmen (z.B. lässt sich ihre Ausführungsdauer angeben). Das zugeordnete Stereotyp ist $<< SA\,Action >>$. Eine SAction wird in ähnlicher Weise auf Basisklassen angewendet, wie die von ihr abgeleitete Response:

SActions werden angewendet auf

- Methoden. In diesem Fall können Eigenschaftswerte vorgegeben werden, die für jede Ausführung einer Methode gelten sollen (z.B. Ausführungszeiten mit *RT duration*). Eine Methode kann durch weitere

Diagramme verfeinert werden (Sequenz-, Kollaboration- oder Aktivitätsdiagramm).

- Nachrichten in einem Sequenz- oder Kollaborationsdiagramm. In diesem Fall sollte die erste *SAction* eine *Response* sein. Danach wird eine Abfolge von *SActions* erwartet. Andere Nachrichten werden ignoriert.
- eine Aktion in einem Aktivitätsdiagramm. In diesem Fall sollte die Aktion ein Nachfolger einer *Response* sein, der wiederum ein *Trigger* vorausgeht. Eine *SAction* kann durch ein untergeordnetes Aktivitätsdiagramm weiter verfeinert werden.

Eine Anwendung auf eine Action Execution ist hier nicht vorgesehen. Wird der Ausführungsbalken an einer Instanz als *Response* gekennzeichnet, ist eine weitere Definition von Einzelschritten schwierig, da Ausführungsbalken nicht das Mittel der Wahl sind, Szenarien zu modellieren. Daher wurde dieser Fall nicht betrachtet.

Eigenschaftswerte. Für jede SAction muss eine maximale Ausführungszeit spezifiziert werden. Es handelt sich dabei um die reine Rechenzeit ohne Unterbrechung oder Blockierung. Diese Rechenzeit wird entweder mit *RT duration* explizit angegeben oder ergibt sich durch untergeordnete Einzelschritte. Sind Verzweigungen in einem Aktivitätsdiagramm spezifiziert, ist im allgemeinen Fall anhand des Modells nicht festzustellen, welche Alternativen zur Laufzeit tatsächlich durchlaufen werden. Insbesondere wenn sich Verzweigungen auf Bedingungen beziehen, die von der Systemumgebung abhängen, sind die tatsächlich möglichen Durchläufe nicht aus dem Modell zu ermitteln. Daher wird in der Implementierung der *zeitlich längste* Pfad als Ausführungszeit bestimmt. Da Bedingungen dann ignoriert werden, führen zyklische Ausführungspfade immer zu unendlichen Ausführungszeiten und sind daher nicht zulässig. Zyklische Ausführungen von *Tasks* sind dagegen sehr wohl durch die Angabe eines entsprechenden Auftrittsmusters für einen Trigger zu modellieren.

4.2.6 Echtzeitmodell

Ausgangspunkt einer Analyse ist eine Beschreibung einer Echtzeitsituation (*Real Time Situation,* vgl. Abbildung 4.4). Sie ist mit den wesentlichen Elementen einer Analyse, den Prozessoren, den Prozessen und den Ressourcen

direkt verbunden. Das SPT-Profil sieht für eine Echtzeitsituation das Stereotyp << *SA Situation* >> vor. Dieses Stereotyp kann entfallen, wenn man davon ausgeht, dass ein gesamtes Modell genau diese Echtzeitsituation beschreibt. In dieser Arbeit wird dieser Standpunkt eingenommen, so dass eine explizite Anwendung von *SA Situation* nicht notwendig ist.

4.2.7 Ressourcen

Ressourcen werden im Domain-Modell durch die Klasse *SResource* berücksichtigt. Der Begriff der allgemeinen *Resource* ist im Profil bereits anderweitig belegt. Eine SResource trägt den besonderen Eigenschaften einer Ressource im Kontext einer Schedulability-Analyse Rechnung. Im UML-Modell wird das Stereotyp << *SA Ressource* >> verwendet. Eine konkrete Ausführung einer SAction (=Folge von Action Executions) kann eine oder mehrere Ressourcen reservieren oder freigeben. Im Domain-Metamodell wird daher ein *AcquireService* und ein *ReleaseService* als Spezialisierung einer Action Execution eingeführt (Abb. 4.3). Im UML-Modell werden das Reservieren und Freigeben von Ressourcen durch die Stereotypen << *GRM acquire* >> und << *GRM release* >>[1] gekennzeichnet. Sie sind mit den entsprechenden SResourcen durch eine Assoziation verbunden. Eine Ressource gilt vom Beginn eines AcquireService bis zum Ende eines ReleaseService als reserviert. Ressourcenzugriffe werden nicht gesondert gekennzeichnet, sondern als eine SAction modelliert, die mit einer Ressource verknüpft ist. Ein Zugriff auf eine nicht reservierte exklusive Ressource ist unzulässig. Bei einer Schedulability-Analyse ist insbesondere der kritische Bereich einer Ressource (vgl. 4.1.2) interessant. Da hier nur die Zeitpunkte der Reservierung und der Freigabe eine Rolle spielen, kann bei einem Ressourcenzugriff auch auf die Verbindung zur Ressource verzichtet werden, solange diese nicht zur Veranschaulichung beibehalten werden soll.

Folgende UML-Modellelemente können als Ressource gekennzeichnet werden:

- Eine Instanz oder Rolle in einem Sequenz- oder Kollaborationsdiagramm.
- Ein Objekt oder eine Klasse.

[1] Diese Stereotypen stammen aus dem Paket General Ressource Modeling.

- Ein Objektzustand in einem Aktivitätsdiagramm.

Die Stereotypen << *GRM acquire* >> und << *GRM release* >> können prinzipiell auf dieselben UML-Elemente angewendet werden, wie das oben beschriebene Stereotyp << *SAAction* >>. Implementiert ist eine Anwendung in Kollaborationsdiagrammen.

Eigenschaftswerte. Für die Reservierung und Freigabe von Ressourcen können folgende Zeitspannen definiert werden: Die *Acquisition Time* definiert die Zeitdauer, die bei der Reservierung anfällt, die *Deacquisition Time* entsprechend die Zeitdauer für die Freigabe. Diese Zeiten werden einer SResource zugeordnet.

Sind in einem System verschiedene Möglichkeiten der Zugriffskontrolle möglich, kann der verwendete Zugriff entweder für jede SResource oder global für eine Execution Engine festgelegt werden. Für Execution Engines wird der Eigenschaftswert *SAAccessPolicy* und für SResourcen der Wert *SAAccessControl* verwendet. Mögliche Ausprägungen sind *PriorityInheritance*, *PriorityCeiling* und *Semaphores*.[2]

Modellverknüpfungen. Für eine Analyse muss die Verknüpfung zwischen einer Zugriffsaktion (Reservierung, Freigabe) und einer Ressource hergestellt werden. Je nach Anwendung der Stereotypen für den Zugriff gibt es folgende Möglichkeiten:

- *Methode.* Alle Aufrufe dieser Methode gelten als Zugriffsaktionen. Es wird erwartet, dass die zugehörige Klasse als Ressource gekennzeichnet ist.
- *Nachricht in einem Sequenz- oder Kollaborationsdiagramm.* Die Instanz, die das Ziel dieser Nachricht ist, ist die zugeordnete Ressource und muss als solche gekennzeichnet werden.

4.2.8 Release Jitter

Im SPT-Profil fehlt bisher ein expliziter Eigenschaftswert zur Modellierung von Release Jittern. Es gibt zwar die Möglichkeit, diese innerhalb des Auf-

[2] Implementiert ist derzeit nur eine Analyse unter Berücksichtigung des Priority-Ceiling-Protocols.

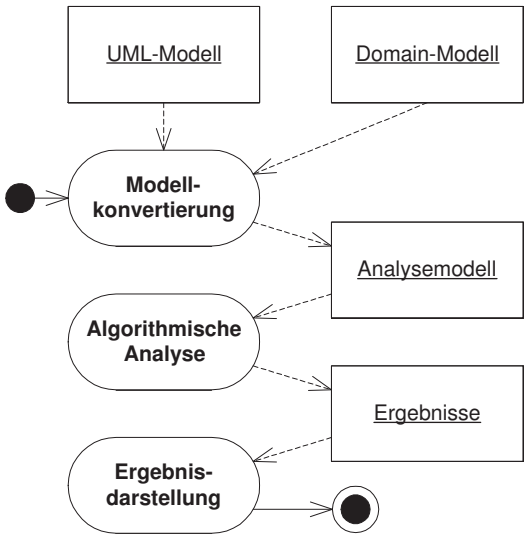


Abbildung 4.7: Ablauf Schedulability-Analyse

trittsmusters eines Triggers zu berücksichtigen. Da es sich um einen wesentlichen Parameter bei der Analyse handelt, wird eine Response um einen entsprechenden Wert *Release Jitter* ergänzt. Auf UML-Ebene kann ein mit $<< SA\,Response >>$ markiertes Element mit dem neu eingeführten Eigenschaftswert $SA\,Jitter$ markiert werden.

4.3 Realisierung der Analyse

Nachdem im Abschnitt zuvor dargestellt wurde, wie sich Scheduling-Aspekte in UML-Modellen spezifizieren lassen, ist dieser Abschnitt der praktischen Durchführung der Analyse gewidmet.

Die eigentliche Scheduling-Analyse wird nicht auf dem UML-Modell, sondern auf einem Analysemodell angewandt. In diesem Modell ist die Information, die für eine Analyse wichtig ist und dem UML-Modell entnommen werden kann, konzentriert aufbereitet. Modellelemente des ursprünglichen UML-Modells, die für eine Analyse des Schedulings nicht relevant sind, werden nicht übernommen. Damit gliedert sich die Scheduling-Analyse in die Phasen Modellkonvertierung und algorithmische Analyse (Abb. 4.7).

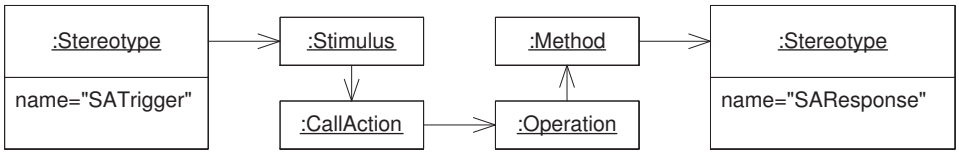


Abbildung 4.8: Rekonstruktion von Objektverbindungen

4.3.1 Modellkonvertierung und Durchführung der Analyse

Das Analysemodell wird auf Basis der Stereotypen und Eigenschaftswerte erstellt, mit denen das Ausgangsmodell versehen ist. Stereotypen repräsentieren Domain-Objekte aus dem konzeptionellen Domain-Modell für die Scheduling-Analyse. Die mit den Stereotypen verbundenen Eigenschaftswerte bestimmen die Attributausprägungen dieser Analyseobjekte.

Verbindungen zwischen diesen Objekten müssen sowohl zu den konzeptionellen Assoziationen im Domain-Modell als auch zu den modellierten Verbindungen des Ausgangsmodells konsistent sein. Die Ableitung von Objektverbindungen im Analysemodell setzt eine genaue Kenntnis des Metamodells von UML voraus, da Stereotypen oft nicht direkt, sondern indirekt über mehrere nicht mit Stereotypen versehene Instanzen verbunden sind. Ein Beispiel dafür wird in Abbildung 4.8 gegeben. Abgebildet sind einige Instanzen von UML-Metaklassen, die einen Teil eines UML-Modells repräsentieren: Ein *Stimulus*, der mit dem Stereotyp $<<SATrigger>>$ versehen ist, führt zum Aufruf einer Methode, die mit dem Stereotyp $<<SAResponse>>$ beschriftet ist. Die Verbindung zwischen Trigger und Response führt über weitere Metamodell-Instanzen (*CallAction, Operation* und *Method*). Je mehr verschiedene Zusammenhänge bei der Modelltransformation berücksichtigt werden, desto mehr Freiheiten hat der Modellierer bei der Anwendung von Stereotypen.

Die Implementierung der Schedulability-Analyse nutzt die Modellverknüpfungen, die in Abschnitt 4.2 beschrieben werden. Die Implementierung realisiert eine Menge von Funktionen, die ausgehend von einem stereotypisierten Modellelement x ein stereotypisiertes Modellelement y finden. Das genaue Vorgehen ist dabei vor allem von der Programmierschnittstelle des Modellierungswerkzeugs abhängig, über welche die Modellinformation abgefragt wird.

Diese Schnittstelle spiegelt die spezifische Datenstruktur des Werkzeugs wieder und daher ist eine Modellkonvertierung nur schwer auf andere Werkzeuge zu übertragen. Die Modelltransformation wird vereinfacht, wenn Folgendes zutrifft:

- Teile des UML-Metamodells finden sich im Domain-Metamodell wieder. In diesem Fall lassen sich Teile der Modellierung in ein Analysemodell übernehmen.
- Es werden Eigenschaftswerte eingeführt, die explizite Verknüpfungen zu anderen Modellelementen enthalten. Beispielsweise kann eine Aktion zur Reservierung einer Ressource einen Verweis auf ein Modellelement enthalten, das diese Ressource modelliert.
- Es werden zusätzliche, redundante Stereotypen und Eigenschaftswerte eingeführt, die eine Modellkonvertierung erleichtern.

Um ein Analysemodell zu erstellen, werden aus einem komplexen UML-Modell zunächst die Teile heraus gefiltert, die für eine Analyse relevant sind. Um diese Objekte des Analysemodells zu ermitteln, wird das gesamte UML-Modell nach Stereotypen durchsucht. Im Folgenden wird ein grober Überblick über das Vorgehen gegeben:

- Die Menge der Prozesse ergibt sich aus den Elementen, die mit dem Stereotyp $<< SA\,Schedulable >>$ gekennzeichnet sind.
- Um einen Task zu ermitteln, der von einem Prozess ausgeführt wird, wird zunächst ein Trigger benötigt. Ist einem Prozess ein Aktivitätsdiagramm untergeordnet, findet sich der Trigger dort. Ansonsten wird ein Trigger in einem Sequenz- oder Kollaborationsdiagramm gesucht. In diesem Fall befindet er sich an einer Nachricht, die von einem Prozess empfangen wird.
- Das Stereotyp $<< SA\,Response >>$ kann auf Methoden angewendet werden. Wird keine Verfeinerung in Einzelschritte in einem Aktivitätsdiagramm oder in Szenarien vorgenommen, muss der Eigenschaftswert *RT duration* mit der maximalen Ausführungszeit besetzt sein. Mit den Eigenschaftswerten *SA Jitter* und *SA AbsDeadline* können ein Jitter bei der Taskauslösung und eine Deadline spezifiziert werden. Wenn als Prioritätsordnung `FixedPriority` gewählt wird, muss zusätzlich für

jeden Task eine Priorität angegeben werden. Dieses passiert über den Eigenschaftswert *SA Priority*.

- In Klassen, Rollen oder Instanzen wird nach << *SA Resource* >> gesucht. Die Reservierung und Freigabe von Ressourcen benötigt oft eine gewisse Zeitdauer, die über die Eigenschaftswerte *SA Acquisition* und *SA Deacquisition* modelliert werden. Fehlt diese Angabe, wird jeweils der Standardwert 0 angenommen.

- << *GRM aquire* >> und << *GRM release* >> modellieren die Reservierung und Freigabe von Ressourcen. Sie werden in Kollaborationsdiagrammen an Nachrichten verwendet. Das Ziel einer solchen Nachricht muss eine << *SA Ressource* >> sein.

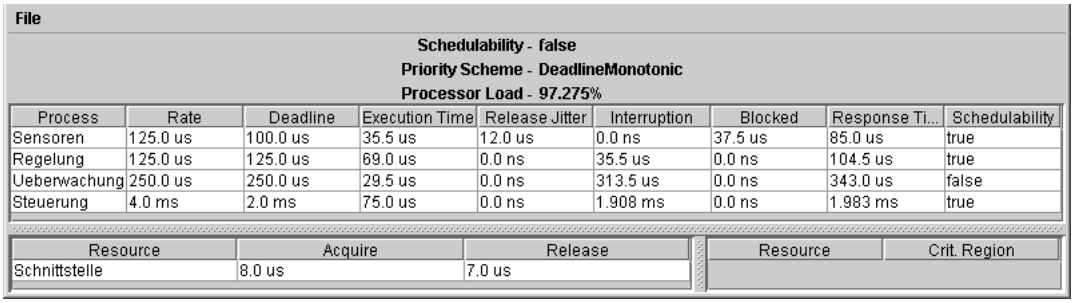
- Die Menge der Hardware-Knoten ergibt sich aus Elementen, die mit << *SA Engine* >> gekennzeichnet sind. Die Prioritätsordnung wird mit dem Eigenschaftswert *SA SchedulingPolicy* wird für alle Prozesse festgelegt. Zulässige Werte sind `RateMonotonic`, `DeadlineMonotonic` und `FixedPriority`.

Als Auftrittsmuster für einen Trigger sind `periodic` oder `bounded` zugelassen. In Kollaborations-, Aktivitäts- und Zustandsdiagrammen können Einzelschritte (Nachrichten, Aktivitäten oder Zustände) mit dem Stereotyp << *SA Action* >> modelliert werden.[3]

In Kollaborationsdiagrammen wird die Reihenfolge der Nachrichten und damit der Aktionen über das Standardschema der Sequenznummern festgestellt, die in UML vorgesehen sind. Bedingungen werden pauschal durch `true` ersetzt und dann der zeitlich längste Pfad zugrunde gelegt. Nebenläufigkeiten sind innerhalb einer Response nicht zugelassen. Bei Aktivitäts- und Zustandsdiagrammen kann die Reihenfolge der Aktionen direkt dem Diagramm entnommen werden. Wie bei Kollaborationsdiagrammen gilt, dass Bedingungen nicht ausgewertet und Nebenläufigkeiten ausgeschlossen werden.

Die oben genannten Diagramme stellen den Ablauf einer einzelnen Taskausführung da. Zyklen innerhalb einer Taskausführung sind nicht zugelassen, da diese potentiell zu unendlichen Wiederholungen von Aktionen mit

[3]Bei Sequenzdiagrammen kann konzeptionell in ähnlicher Weise verfahren werden; allerdings ist die Reihenfolge von Nachrichten bisher nicht in ausreichender Weise in der Datenstruktur des verwendeten Werkzeugs (Poseidon) abgelegt. Daher wurde eine Zerlegung in Einzelschritte bisher nicht für Sequenzdiagramme implementiert.



File

Schedulability - false
Priority Scheme - DeadlineMonotonic
Processor Load - 97.275%

Process	Rate	Deadline	Execution Time	Release Jitter	Interruption	Blocked	Response Ti...	Schedulability
Sensoren	125.0 us	100.0 us	35.5 us	12.0 us	0.0 ns	37.5 us	85.0 us	true
Regelung	125.0 us	125.0 us	69.0 us	0.0 ns	35.5 us	0.0 ns	104.5 us	true
Ueberwachung	250.0 us	250.0 us	29.5 us	0.0 ns	313.5 us	0.0 ns	343.0 us	false
Steuerung	4.0 ms	2.0 ms	75.0 us	0.0 ns	1.908 ms	0.0 ns	1.983 ms	true

Resource	Acquire	Release
Schnittstelle	8.0 us	7.0 us

Resource	Crit. Region

Abbildung 4.9: Screenshot tabellarische Ausgabe

unbeschränkter Zeitdauer führen würden. In diesem Fall ergäbe sich trivialerweise, dass keine Deadline eingehalten werden kann. Die wiederholte Ausführung eines Tasks wird durch Angabe eines *SA Occurence*-Musters modelliert.

Durchführung der Analyse. Das Analyseprogramm verwendet den Algorithmus aus 4.1.2. Damit sind sich überlappende Aktivierungen von Task möglich, wenn $D_i > T_i$. Es können exklusive Ressourcen modelliert werden, deren Zugriffe nach dem Priority-Ceiling-Protocol verwaltet werden. Weiterhin werden Release Jitter J_i berücksichtigt.

4.3.2 Realisierung

Die in diesem Kapitel vorgestellten Erweiterungen und darauf aufsetzenden Analysen sind als Programm in Java mit ca. 100 Klassen realisiert [DHK03] und als Plug-In in das Werkzeug *Poseidon for UML* integriert. Poseidon qualifiziert sich für die Erweiterung durch eine offene Schnittstelle für Plug-Ins und eine gute Unterstützung von Stereotypen und Eigenschaftswerten.

Die Analyse setzt auf *XMI*, dem Standardaustauschformat für UML, auf[4]. Ein in diesem Format abgespeichertes Modell wird eingelesen, analysiert und das Ergebnis in tabellarischer Form ausgegeben (Abb. 4.9). Eine Funktion zum erneuten Laden erlaubt die schnelle Analyse von unterschiedlichen Parameterkombinationen. Die Ausgabe beinhaltet folgende Informationen:

[4]Da XMI Standard ist, könnte man vermuten, dass sich auch andere Werkzeuge einsetzen lassen, die in dieses Format importieren. Das ist leider nicht so, da jedes Werkzeug seinen eigenen XMI-Dialekt hat.

Schedulability Das Gesamtergebnis der Analyse. Sie ist `true`, wenn alle Tasks stets ihre Deadlines einhalten und ansonsten `false`.

Priority Scheme Das Prioritätenschema wird *SASchedulingPolicy* von *<< SAEngine >>* entnommen.

Processor Load Hier wird die rechnerische Prozessorauslastung angegeben.

Danach folgt eine Liste aller Tasks im System. Dabei werden in den ersten fünf Spalten Parameter ausgegeben, die dem Modell entnommen werden und in den letzten vier Spalten daraus abgeleitete Ergebnisse.

Process Name des ausführenden Prozesses. Es wird der Name des Modellelementes ausgegeben, das mit *<< SASchedulable >>* gekennzeichnet wird.

Rate Gibt die Periode T_i bei zyklischen Tasks oder die minimale Zeit zwischen zwei Aktivierungen bei aperiodischen Tasks an.[5]

Deadline Die Deadline D_i eines Tasks.

Execution Time Die maximale Ausführungszeit C_i eines Tasks, der weder unterbrochen noch blockiert wird.

Release Jitter Der Release Jitter J_i.

Aus diesen Angaben werden folgende Werte abgeleitet. Dabei werden die Algorithmen aus Abschnitt 4.1.2 verwendet.

Interruption Die maximale Unterbrechungszeit I_i durch andere Tasks höherer Priorität.

Blocked Die maximale Blockierung B_i aufgrund belegter Ressourcen.

Response Time Die maximale Antwortzeit R_i einschließlich der Ausführungszeit C_i, der Unterbrechungszeit I_i und der Blockierzeit B_i.

[5] Diese Angabe wird dem *SAOccurence*-Muster der Trigger entnommen. Modellierte Zyklen in Kollaborationsdiagrammen sind wie oben beschrieben nicht erlaubt.

Schedulability Gibt an, ob ein Task innerhalb seiner Deadline beendet wird. Sie ist genau dann gegeben, wenn $R_i \leq D_i$.

Bei der Interpretation der Werte ist zu beachten, dass im Falle der Verletzung einer Deadline der Algorithmus aus 4.1.2 vorzeitig abbricht, d.h. die berechnete Antwortzeit stellt dann nur eine untere Grenze da.

Im Feld unten links wird eine Liste von Ressourcen mit ihrer jeweiligen Reservierungs- und Freigabezeit (Eigenschaftswerte *SA Acquisition* und *SA Deacquisition* von << *SA Resource* >>) angegeben. Das Feld rechts gibt die Dauer des längsten kritischen Bereichs aller benutzen Ressourcen an. Es bezieht sich jeweils auf denjenigen Task, der gerade im Feld darüber selektiert ist.

4.3.3 Beispiel

Um die Anwendung der beschriebenen Stereotypen in einem Gesamtmodell zu verdeutlichen, wird im Folgenden die vereinfachte Modellierung einer Robotersteuerung dargestellt.[6]

In Abbildung 4.10 wird in einem Klassendiagramm eine Übersicht über Prozesse und Ressourcen gegeben. Es sind dabei folgende Bestandteile beteiligt:

- *Industrial Automation Protocol (IAP).* Dieses Protokoll sichert die Hochgeschwindigkeitskommunikation mit Sensoren und Motoren. Hier ist streng genommen nicht das Protokoll selbst gemeint, sondern die Software-Komponente, die auf Steuerungsseite auf den Kommunikationsbus zugreift. Aus Steuerungssicht agiert sie wie eine exklusive Ressource, da zu einem bestimmten Zeitpunkt nur ein Prozess auf den Bus zugreifen kann. Für eine asynchrone Kommunikation muss zuvor Bandbreite reserviert werden.

- *Sensoren.* In dieser Klasse werden Sensordaten verwaltet. Die Sensorwerte werden über den Datenbus per IAP übermittelt. Die Klasse stellt Methoden zum Abfragen der Sensorwerte für andere Steuerungskomponenten (Regelung, Überwachung) bereit. Es handelt sich allerdings *nicht* um eine aktive Klasse.

[6] Die Beispielabbildungen wurden mit dem UML-Werkzeug Poseidon erstellt, für das die Analyse als Plug-In implementiert wurde.

- *Aktoren.* Diese Klasse ist das Gegenstück zu Sensoren. Das heißt, hier werden Daten zur Ansteuerung der Motoren aufbereitet, bevor sie über das IAP verschickt werden. Auch hierbei handelt es sich um eine passive Klasse.

- *Bahnplanung.* Hierbei handelt es sich um eine aktive Klasse, d.h. um einen Prozess.[7] Er benötigt Rechenzeit, um aus Anwenderbefehlen[8] kollisionsfreie Bahnen zu berechnen. Diese werden in Form von Stützstellen der Regelung übergeben.

- *Regelung.* Hierbei handelt es sich um einen zentralen Prozess, der Stützstellen von der Bahnplanung auf Basis der aktuellen Sensorwerte in elementare Motorbefehle umsetzt.

- *Überwachung.* Die Überwachung ist Teil des Sicherheitskonzeptes und verwaltet Abbruchbedingungen von elementaren Benutzerbefehlen.

- *Transformation.* In dieser Klasse werden Umrechnungsalgorithmen gekapselt, die von der Roboterstruktur abhängen. Typische Beispiele sind die Umrechnung der Position des Endeffektors (Greifers) des Roboter in Motorkoordinaten und deren Umkehrung.

Die Aufteilung der Steuerung in Prozesse ist im Verteilungsdiagramm der Abbildung 4.11 zu sehen. Das Steuerungssystem besteht aus insgesamt vier Prozessen, die mit unterschiedlicher Priorität ablaufen. Es handelt sich dabei um Instanzen der Klassen, die in Abbildung 4.10 als $<< SA\,Schedulable >>$ gekennzeichnet sind. In diesem Diagramm werden sie einem Hardware-Knoten zugeordnet. Für diesen Knoten wird wiederum das Schema angegeben, das zur Prioritätenverteilung herangezogen wird. In diesem Fall werden Prioritäten gemäß der Deadline eines Tasks verteilt.

In Abbildung 4.10 ist die Methode *control*() der Regelung als Response markiert. In Abbildung 4.12 ist ein Aktivitätsdiagramm abgebildet, das einen getriggerten Aufruf von *Regelung.control*() verfeinert darstellt. Der Regeltakt wird durch den verwendeten Bus vorgegeben: Alle $125\mu s$ benötigen die

[7] Die übliche Darstellung von aktiven Klassen durch einen breiteren Rand wird von Poseidon nicht unterstützt.

[8] Es handelt sich hierbei um so genannte Aktionsprimitive, deren Realisierung ein zentrales Thema des SFB 562 ist. Details sind in [Fin04] ausgearbeitet.

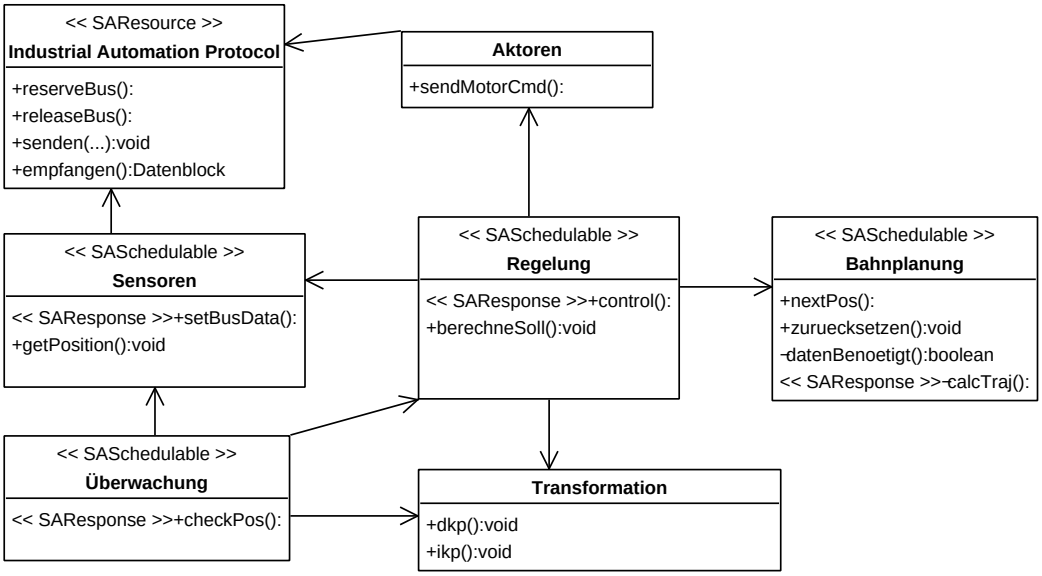


Abbildung 4.10: Klassendiagramm: Prozesse und Ressourcen

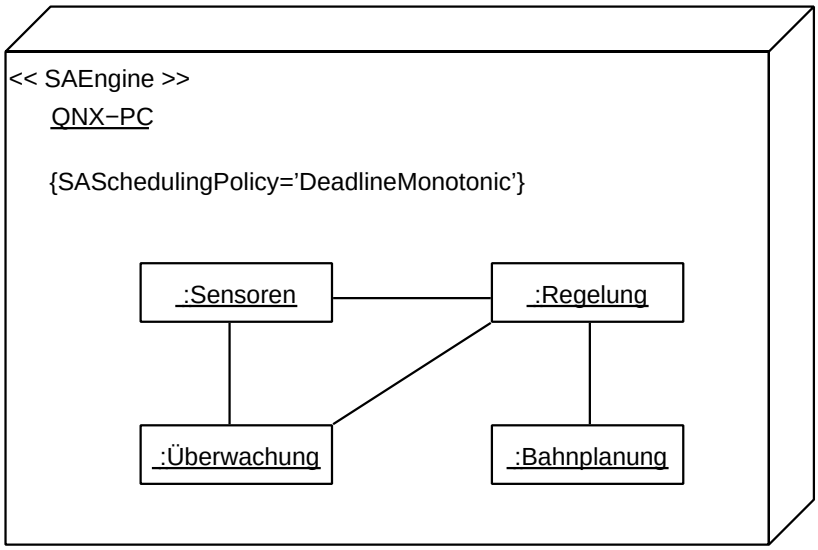


Abbildung 4.11: Verteilungsdiagramm: Prozesse und Scheduling-Verfahren

Motoren eine neue Vorgabe der Regelung. Dabei werden folgende Schritte ausgeführt:

Es werden aktuelle Sensorwerte (Gelenkstellungen) mittels *Sensoren.get Position*() abgefragt und danach in kartesische Koordinaten umgerechnet ($Transformation.dkp(istGelenk)$). Dabei werden strukturabhängige Algorithmen verwendet - zunächst zur Lösung des *direkten kinematischen Problems (DKP)* (Ableitung der kartesischen Raumkoordinaten aus den Gelenkkoordinaten) und dann des *inversen kinematischen Problems (IKP)* (Bestimmung der Gelenkkoordinaten aus einer kartesischen Position). Es überprüft, ob die Ist-Werte der Sensoren eine Anpassung der Soll-Werte erfordern. Wenn dieses der Fall ist, werden neue kartesische Soll-Werte berechnet, diese in Gelenkpositionen ($Transformation.ikp(sollKoord)$) umgewandelt und mit $Aktoren.send\ MotorCmd(sollGelenk)$ zur Übertragung vorbereitet. Ansonsten bleiben die Soll-Werte unverändert.

Jeder Einzelschritt ist mit einer Ausführungszeit versehen (Eigenschaftswert *RT duration*). Da nicht vorauszusehen ist, ob zur Laufzeit eine Anpassung des Soll-Wertes vorgenommen werden muss, wird für den gesamten Ablauf der Pfad mit der längsten Ausführungszeit zu Grunde gelegt.

Die Modellierung mit einem Kollaborationsdiagramm ist in Abbildung 4.13 dargestellt. Es handelt sich dabei um die Response *setBusData*() von *Sensoren*. Der Trigger wird von einer Hardware-Komponente periodisch ausgelöst: Ein *Circle Start Telegram* (CST) markiert den Beginn eines Regelzyklus. Nachdem der Trigger dieses Tasks stattgefunden hat, werden von *Sensoren* Werte von der Buskomponente IAP gelesen.

Der Buszugriff über das IAP wird in diesem Beispiel als eine exklusive Ressource betrachtet. Insbesondere für asynchrone Zugriffe muss dafür zunächst Bandbreite reserviert werden. In Abbildung 4.10 ist die entsprechende Instanz mit $<< SA\,Ressource >>$ gekennzeichnet worden. Sie stellt in ihrer Schnittstelle die zwei Methoden *reserveBus*() und *releaseBus*() für Reservierung und Freigabe bereit. Die Zeiten für Reservierung und Freigabe können in Form der Eigenschaftswerte *SA Acquisition* und *SA Deacquisition* beschrieben werden. Die entsprechenden Zeiten gelten für jeden modellierten Aufruf dieser Methoden.

In Abbildung 4.13 wird daher zunächst diese Komponente reserviert. Die Reservierung ist durch das entsprechende Stereotyp gekennzeichnet und kann zugeordnet werden, da das Ziel der Nachricht wie gefordert eine Ressource ist (vgl. Abb. 4.10). Die Zeit für die Reservierung ist ggf. bei der Ressource

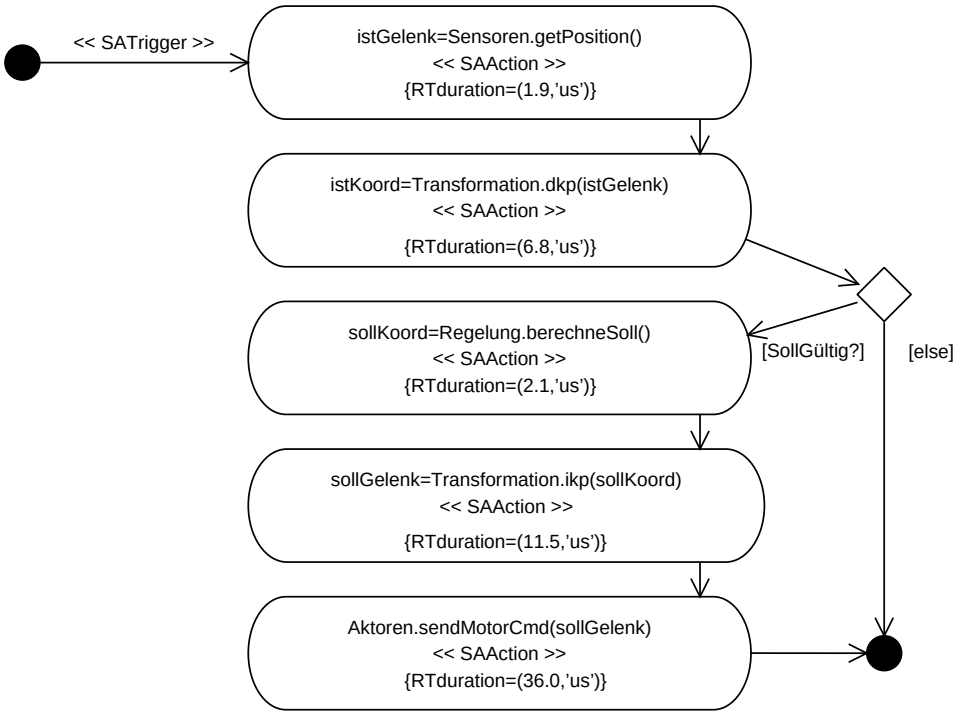


Abbildung 4.12: Aktivitätsdiagramm

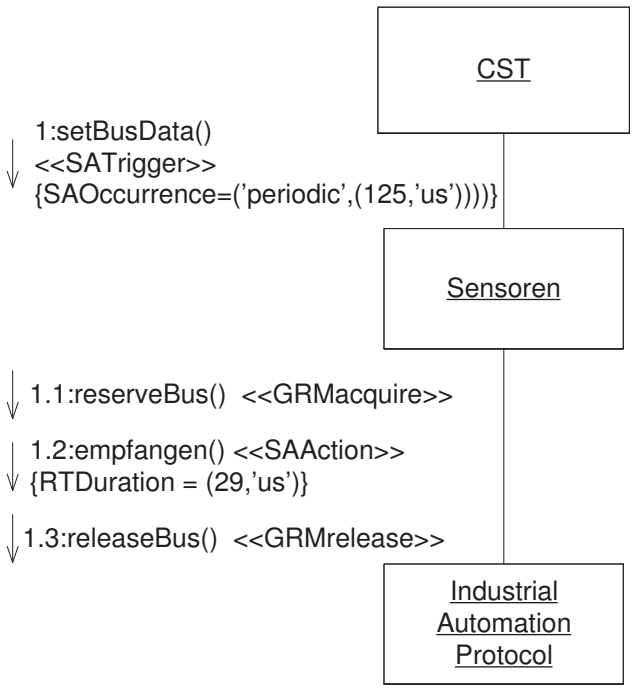


Abbildung 4.13: Kollaborationsdiagramm mit Echtzeitannotationen

eingetragen. Sie wird ebenso berücksichtigt, wie die explizit angegebene Dauer der nachfolgenden Aktion. Abgeschlossen wird der Task mit der Freigabe der Ressource.

Kapitel 5

Zusammenfassung und Ausblick

Zielsetzung dieser Arbeit war, den Entwicklungsprozess von Echtzeitsystemen zu verbessern. Als Ansatzpunkt dazu wurde die Modellebene gewählt, da damit in frühen Phasen der Software-Entwicklung angesetzt wird. Fehler, die zu Beginn der Entwicklung übersehen werden, verursachen erfahrungsgemäß besonders hohe Kosten, wenn sie später behoben werden.

Als Schnittstelle zum Anwender wird die UML genutzt. Dieses bietet sich an, da die UML im Bereich der Modellierungssprachen eine herausragende Position einnimmt. Ein wichtiger Faktor ist die große Anzahl von UML-Modellierungswerkzeugen, die zum Teil erweiterbar gehalten sind. Die Verfahren, die hier entwickelt wurden, sind so konzipiert, dass sie in diese Werkzeuge integrierbar sind. Dies wurde durch prototypische Implementierungen für ein kommerzielles Werkzeug nachgewiesen.

Der Schwerpunkt dieser Arbeit lag darin, formale Techniken praxisnah und effizient einsetzbar zu machen. Ein entscheidendes Kriterium für das Erreichen dieser Zielsetzung ist, dass der zusätzliche Aufwand des Benutzers so gering wie möglich gehalten wird. Theorienahe Werkzeuge leisten oft Erstaunliches, werden aber in der Praxis nur selten angewandt. Gerade für Echtzeitsysteme bietet sich der Einsatz von diesen Werkzeugen an, da deren zeitlich-dynamisches Verhalten eine Komplexität aufweist, die oft weder getestet noch mit dem menschlichen Verstand nachvollzogen werden kann. Model-Checker für Zeitautomaten analysieren gerade dieses Verhalten mit hochoptimierten Algorithmen. Dass sie trotzdem im industriellen Umfeld kaum eingesetzt werden, liegt daran, dass die Modellierung mit Zeitautomaten überaus zeitaufwendig ist.

Daraus ergibt sich die zentrale Frage, die diese Arbeit zu beantworten

versucht: Wie lassen sich formale Techniken zur automatischen Analyse von Echtzeitsystemen einsetzen, die mit UML modelliert sind?

Dazu wurden folgende notwendige Schritte identifiziert:

1. Es wird eine Modellierungssprache benötigt, mit der ein Anwender konkrete Problemsituationen beschreiben kann. Die UML liefert dazu nur den Rahmen, in dem eine problemspezifische Sprache entwickelt wird (Modellebene).

2. Passend zu einer Problemstellung wird ein Verfahren ausgewählt, mit dem dieses Problem gelöst werden kann. Dazu wurde evaluiert, welches Verfahren für welches Problem geeignet ist (Analyseebene).

3. Es wird ein Werkzeug benötigt, das ein vom Anwender erstelltes Modell automatisch in eine Problembeschreibung des gewählten Verfahrens übersetzt. Da sich die Modellwelt des Anwenders und des formalen Werkzeugs stark unterscheiden können, kann dieser Übergang komplex sein (Transformation).

Dieses Grundschema zieht sich durch diese Arbeit. Zu beachten ist, dass die problemspezifische Sprache *nicht* die UML ist. In ihrer Gesamtheit ist die UML zu mächtig und semantisch zu unpräzise, um formal analysierbar zu sein. Sie liefert die Basis, auf der problemspezifische Sprachen definiert werden. Dabei wird, wie zum Beispiel bei der Schedulability-Analyse, von den UML-eigenen Erweiterungsmechanismen Gebrauch gemacht.

Um der Komplexität des Entwurfs von Echtzeitsystemen gerecht zu werden, muss ein Entwicklungsprozess mehrere Abstraktionsebenen durchlaufen, die wiederum verschiedene Sichten auf das Softwaresystem haben. Entsprechend ist eine automatische Analyse am wirkungsvollsten, wenn sie auf verschiedenen Ebenen ansetzt und mehrere Sichten berücksichtigt. Allerdings ist für ein bestimmtes Problem nicht jede Sicht relevant. In dieser Arbeit lag der Fokus auf dem Echtzeitverhalten von Software. Daher waren vor allem Sichten interessant, welche die Software-Dynamik und Zeitaspekte widerspiegeln. Im Folgenden werden die gewählten Ansätze zusammengefasst und von einander abgegrenzt.

Im ersten Ansatz wurde die Verifikation von Echtzeitmodellen in relativ früher Phase behandelt. Dieser Ansatz zielt darauf ab, dass der Übergang

von einer Anforderungsspezifikation hin zu einem Entwurfsmodell der Implementierung verifizierbar wird. Gerade in dieser frühen Phase sollten Modellierungsfehler vermieden werden, da deren Beseitigung in späten Phasen umso aufwendiger wird. Das besondere Augenmerk dieses Ansatzes liegt auf der dynamischen Modellierung einschließlich von Zeitaspekten. Sequenzdiagramme repräsentieren Anforderungen, Zustandsdiagramme den Systementwurf.

Da dieser Ansatz in einer frühen Phase ansetzt, lässt sich eine Modellsprache verwenden, die nahe am UML-Kern bleibt. Allerdings zeigte sich, dass auch hier die Einführung zusätzlicher Konstrukte nicht ganz zu vermeiden war. Beispielsweise musste der Status eines Szenarios definiert werden. Es macht für eine Analyse einen großen Unterschied, ob eine definierte Abfolge von Ereignissen die einzig mögliche ist oder ob sie nur einen Ausführungspfad darstellt. Da die UML keine derartigen Konstrukte vorsieht, musste der UML-Standard an dieser Stelle erweitert werden. Ansonsten wurden die Änderungen so moderat wie möglich gehalten und eine Semantik umgesetzt, die sich an die Vorgaben des Standards hält und implementierungsnahe Detailaspekte zunächst unberücksichtigt lässt.

Als Verfahren zur Verifikation wurde Model-Checking auf Basis von Zeitautomaten ausgewählt. Vorteilhaft war dabei, dass sich eine hochentwickelte Verifikationstechnologie nutzen ließ. Allerdings ist die Transformation von UML-Modellen in Zeitautomaten aufwendig, da sich einige Konstrukte der UML-Welt mit Zeitautomaten nur umständlich darstellen lassen. An einigen Stellen mussten Kompromisse eingegangen werden, um die Umsetzung in der Größe überschaubar zu halten und eine Verifikation von realen Modellen zu ermöglichen.

Als Ergebnis wurde ein Werkzeug implementiert, das in eine kommerzielle UML-Software integriert wurde. Die eigentliche Verifikation läuft auf Knopfdruck ab, so dass alle Transformationsschritte dem Anwender verborgen bleiben. Die Ausgabe der Ergebnisse erfolgt wiederum innerhalb dieses Werkzeugs. Die Anwendung von formaler Echtzeit-Verifikation aus einem UML-Werkzeug heraus macht das Innovative dieses Ansatzes aus.

Folgende Veröffentlichungen sind in diesem Zusammenhang entstanden: In [DGV02] wurde in einer Fallstudie die Verifikation eines Systems von Zustandsdiagrammen vorgenommen. Die verallgemeinerten technischen Details der Transformation wurden in [DGH02] geliefert. Eine Übertragung auf eine Statemate-Semantik vor dem Hintergrund einer Fallstudie aus dem Automotive-Bereich wird in [DM02] diskutiert. Die Grundlagen der Transformation von Sequenzdiagrammen wurden in [FHD+99] gelegt. Dieser dort be-

schriebene Ansatz wurde umfangreich erweitert. Die wichtigste Änderung ist, dass mittlerweile eine Verbindung zur Abbildung von Zustandsdiagrammen aus [DGH02] entwickelt wurde. Das erforderte ein Einbeziehen von asynchroner Kommunikation und damit eine völlige Überarbeitung von [FHD+99]. Die Ergebnisse wurden in dieser Arbeit präsentiert.

Dieser Ansatz ist begrenzt, wenn es darum geht, plattformspezifische Besonderheiten zu berücksichtigen. Beispielsweise verfügen Echtzeitbetriebssysteme wie QNX über spezielle Kommunikationsmechanismen, die sich mit den Mechanismen der UML nur schwer in Einklang bringen lassen. Ein Message-Passing von QNX wird aus naheliegenden Gründen im Standarddokument von UML nicht berücksichtigt. Derartige Konstrukte lassen sich mit der UML nachmodellieren - und verursachen so einen gewaltigen Overhead. Sinnvoller ist, solche Konstrukte ohne Umweg über die UML in Zeitautomaten abzubilden.

Aus dieser Motivation entstand ein zweiter Ansatz. Bei der Analyse einer Dynamikmodellierung von QNX stand der Anwendungsbereich im Zentrum des Interesses. Von hier aus wurden die zentralen Konstrukte ermittelt, die für eine Modellierung von QNX-Anwendungen wichtig sind, um die Voraussetzung für eine maßgeschneiderte Analyse zu realisieren. Die Modellierungssprache bleibt weiterhin die UML, die allerdings um benötigte Konstrukte ergänzt wurde. Diese Erweiterungen wurden zu einem neuen Profil zusammengefasst. Anders als im ersten Ansatz spielte hier allerdings die UML-Semantik bei der Analyse der zeitlichen Dynamik keine Rolle. Maßgeblich war allein die implizierte Semantik aus dem Anwendungsbereich. Schließlich sollten die Analyseergebnisse hier auch ihre Gültigkeit haben.

Auch in diesem Ansatz werden die Modelle in Zeitautomaten überführt und mit Hilfe von Model-Checking verifiziert. Es konnte damit nachgewiesen werden, dass die Besonderheiten eines Echtzeitbetriebssystems bei der Modellierung in UML berücksichtigt werden können und formal verifizierbar sind, ohne dass der Anwender Kenntnisse über die Details der zugrunde liegenden Formalismen haben muss. Damit füllt dieser Ansatz eine Lücke in der Literatur, in der vor allem die Verifikation UML-spezifischer Konstrukte im Vordergrund steht. Die Einbeziehung anwendungsnaher Konstrukte ergänzt diese Ansätze auf pragmatische Weise.

Die beiden bisherigen Ansätze realisieren eine dynamische Analyse, d.h. konkrete dynamische Ausführungen werden durch den Aufbau eines Zustandsraums berücksichtigt. Der Rechenaufwand dieser Analyse ist enorm

und kann bei großen Modellen den Rahmen des Machbaren sprengen. In diesem Fall kann es sinnvoll sein, statische Analyseverfahren anzuwenden, deren Rechenaufwand nahezu zu vernachlässigen ist. Diese Verfahren sind etwas gröber als Verfahren, welche die dynamischen Ausführungen direkt untersuchen, liefern aber oft auch die gewünschten Analyseergebnisse. Insbesondere lassen sich maximale Antwortzeiten sowohl mit Hilfe der oben beschriebenen Methoden dynamisch, aber auch statisch analysieren. Eine statische Antwortzeitenanalyse ist wesentlich performanter, arbeitet allerdings in der Regel mit pessimistischen Abschätzungen.

In dieser Arbeit wurde daher in einem dritten Ansatz untersucht, inwieweit sich Verfahren zu Schedulability-Analyse in den UML-Kontext einbinden lassen. Hier stellt sich vor allem die Frage, wie die relevanten Informationen für eine Schedulability-Analyse modelliert werden können. Einen ersten Hinweis liefert das UML-Profil für Schedulability, Performance und Zeit. Es liefert einen Katalog standardisierter Notationen zu diesem Thema. Es enthält aber keine Hinweise für eine konkrete Umsetzung, d.h. es beantwortet nicht die Frage, welche Stereotypen für eine bestimmte Analyseaufgabe benötigt werden, auf welche Modellelemente diese in diesem Zusammenhang sinnvollerweise angewandt werden, wie sie verknüpft sein sollten und wie diese Elemente dann durch einen geeigneten Algorithmus ausgewertet werden können.

Diese Lücke wird in dieser Arbeit geschlossen. Mit [DHK03] wurde nicht nur die praktische Anwendbarkeit des Profils nachgewiesen, sondern auch das wohl erste Werkzeug präsentiert, das eine Schedulability-Analyse von UML-Modellen auf Basis dieses Profils implementiert. Die Realisierung erlaubt eine flexible Verwendung von Stereotypen, d.h. der Modellierer ist weitgehend frei in der Entscheidung, in welchen Modellsichten er die Stereotypen verwendet. Zeitliche Parameter werden in Form von Eigenschaftswerten eingegeben. Auf Knopfdruck wird die Analyse gestartet und ein Ergebnis in Tabellenform geliefert. Diese unmittelbare Rückmeldung erleichtert das Durchspielen mehrerer Modellierungsvarianten.

Ansatz	UML-Analyse	QNX-Analyse	Sched.-Analyse
Abstraktion	Grob	Fein	Mittel
Sichten	2	2	viele
Verfahren	dynamisch	dynamisch	statisch
Analyseaufwand	hoch	hoch	gering
Plattformabhängigkeit	unabhängig	spezifisch	unabhängig

In der Tabelle oben werden die Eigenschaften der Verfahren zusammengefasst. Alle Ansätze verbindet, dass formale Methoden in einer anwenderfreundlichen, praxisorientierten Weise dem Benutzer zur Verfügung gestellt werden. Teils waren dafür komplexe Transformationen nötig, teils mussten aufwendige Modellierungssprachen auf UML-Basis entwickelt werden.

Allerdings unterscheiden sich die Ansätze, wie oben beschrieben, in ihrer Zielsetzung. Wie lassen sich die Werkzeuge also am sinnvollsten in einen Entwicklungsprozess integrieren? In knapper Form zusammengefasst wird folgendes Vorgehen zum Einsatz der Werkzeuge vorgeschlagen:

Am Anfang der Entwicklung wird die Software-Dynamik zunächst losgelöst von einer konkreten Plattform modelliert. Hier liefert das Tool, das Anforderungen in Form von Sequenzdiagrammen und die Systemmodellierung in Form von Zustandsdiagrammen auf Konsistenz überprüft, wertvolle Dienste. Danach sollte ein Gesamtmodell der Software erstellt werden. In dieser Phase werden für das Scheduling relevante Teile des Modells mit Stereotypen markiert. Während der Modellierung kann sofort geprüft werden, ob das System seine Deadlines einhält. Gegebenenfalls werden Umstrukturierungen sofort vorgenommen. Liefert eine Schedulability-Analyse kein Ergebnis, da die zu treffenden Abschätzungen zu pessimistisch sind, kann versucht werden, ein Modell näher an der Implementierung zu erstellen und eine QNX-spezifische Verifikation darauf anzuwenden. Ebenso lassen sich kritische Teilsysteme vertieft analysieren.

Ausblick. Diese Arbeit behandelt nur eine kleine Auswahl der denkbaren Anwendungsmöglichkeiten formaler Techniken zur Qualitätsverbesserung von Software-Modellen. Folgende Möglichkeiten zur Erweiterung dieser Ansätze sind denkbar:

- Einbeziehung neuer Modellelemente/Diagramme.
- Anwendung anderer formaler Methoden oder Werkzeuge.
- Untersuchung neuer Anwendungsbereiche.

Der erste Punkt wird besonders in der UML 2 interessant, da es hier erlaubt ist, dynamische Diagramme beinahe beliebig zu schachteln. Diese Anwendungsmöglichkeiten ergeben eine schwer zu überschauende Menge von Abläufen. Gerade in diesem Fall sollte Model-Checking noch am ehesten dazu in der Lage sein, die möglichen Ablaufszenarien zu untersuchen. Andererseits

dürfte die Komplexität der Abbildung in formale Sprachen sehr hoch sein. Inwieweit sich in diesem Bereich trotzdem gute Ergebnisse erzielen lassen, ist derzeit schwer abzuschätzen, zumal die Entwicklung einer integrierten Semantik längst noch nicht abgeschlossen ist.

Bezüglich des zweiten Punktes ist beispielsweise die Einbeziehung anderer Werkzeuge denkbar. Aktivitätsdiagramme werden in der UML 2 Petri-Netzen ähnlicher. Da läge es nahe, Analysetools für Petri-Netze in UML-Werkzeuge zu integrieren und mit den bestehenden Ansätzen zu verbinden. Weiterhin sollten diskrete Model-Checker ohne Zeit in einigen Bereichen den Analysehorizont erweitern. Die Liste ließe sich noch beliebig ergänzen.

Was den letzten Punkt betrifft, so wäre es denkbar, dass für bestimmte Branchen spezialisierte Profile entwickelt werden, die besser den jeweiligen Anwendungsbereichen angepasst sind. Die Semantik der UML kann nur den kleinsten gemeinsamen Nenner aller Anwendungsbereiche darstellen. Speziellere Anforderungen können nur durch UML-Profile erfüllt werden. Beispielsweise gibt es Bemühungen zur Entwicklung von speziellen Profilen im Automotive-Bereich. Auch hier werden sich interessante Anwendungsgebiete für formale Methoden bieten.

Eigene Veröffentlichungen

Diethers, Finkemeyer, Kohn
Middleware zur Realisierung offener Steuerungssoftware für hochdynamische Prozesse. it - Information Technology, Oldenbourg Verlag, S. 39-47, 01/2004.

Steiner, Diethers, Mücke, Goltz, Huhn
Rigorous Tool-supported Software Development of a Robot Control System. In Proceedings of the 2. Int. Colloquium of the Collaborative Research Centre 562, Braunschweig, 2005.

Huhn, Mutz, Diethers, Florentz, Daginnus
Applications of Static Analysis on UML Models in the Automotive Domain. FORMS/FORMAT (Formal Methods for Automation and Safety in Railway and Automotive Systems), Braunschweig, 2004.

Kohn, Kolbus, Diethers, et. al.
PROSA - A Generic Control Architecture for Parallel Robots. Mechatronics & Robotics, Aachen, 2004.

Diethers, Huhn
Vooduu: Verification of Object-oriented Designs using UPPAAL. Tools and Algorithms for the Construction and Analysis of Systems, Barcelona 2004.

Diethers, Huhn, Kasiuk
Ein Werkzeug zur automatischen Schedulability-Analyse auf Basis von UML-Modellen. Dresden Workshop Circuit and System Design (DASS) and Workshop System Design Automation (SDA), Dresden 2003.

Diethers, Firley, Kröger, Thomas
A New Framework for Sensor Based Robot Programming and Verification. IEEE International Conference on Advanced Robotics, Coimbra, Portugal 2003.

Diethers, Goltz, Huhn
Model Checking UML Statecharts with Time. UML 2002, Workshop on Critical Systems Development with UML, Dresden, 2002.

Diethers, Goltz, Vocke
Analysis of Real-Time Systems Modeled by UML-Statecharts. In Proceedings of the 1. Int. Colloquium of the Collaborative Research Centre 562, Braunschweig, 2002.

Diethers, Mutz
Improving Software Development in the Automotive Area through Tool Supported Modelling and Formal Analysis. System Design Automation (SDA), Pirna, 2002.

Diethers
Tool-Based Analysis of Timed Sequence Diagrams. Informatik-Bericht Nr. 2001-01, TU Braunschweig, 2001.

Firley, Huhn, Diethers, Gehrke, Goltz
Timed Sequence Diagrams and Tool-Based Analysis - A Case Study. The Second International Conference on The Unified Modeling Language, Beyond the Standard (UML'99), Fort Collins, USA, 1999.

Literaturverzeichnis

[AB98] N. Audsley und A. Burns. On fixed priority scheduling, offsets and co-prime task periods. *Information Processing Letters*, 67:65–69, 1998.

[ABJN99] Parosh Aziz Abdulla, Ahmed Bouajjani, Bengt Jonsson und Marcus Nilsson. Handling global conditions in parameterized system verification. In *Computer Aided Verification*, Seiten 134–145, 1999.

[AD94] Rajeev Alur und David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[AD96] R. Alur und D. Dill. Automata-theoretic verification of real-time systems, 1996.

[AEY00] Rajeev Alur, Kousha Etessami und Mihalis Yannakakis. Inference of message sequence charts. In *International Conference on Software Engineering*, Seiten 304–313, 2000.

[AFM+02] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson und Wang Yi. Times - a tool for modelling and implementation of embedded systems. In *Proceedings of 8th International Conference, TACAS 2002*, Band 2280 aus *LNCS*, Seiten 460–464. Springer-Verlag, 2002.

[AFM+03] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson und Wang Yi. Times: a tool for schedulability analysis and code generation of real-time systems. In *Proceedings of the 1st International Workshop on Formal Modeling and Analysis of Timed Systems*, 2003.

[AGS00] Karine Altisen, Gregor Gler und Joseph Sifakis. A methodology for the construction of scheduled systems. In *Proceedings of the 6th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, Seiten 106–120. Springer-Verlag, 2000.

[AHP96] Rajeev Alur, Gerard J. Holzmann und Doron Peled. An analyzer for Message Sequence Charts. In Tiziana Margaria und Bernhard Steffen (Hrsg.), *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, Band 1055 aus *LNCS*, Seiten 35–48. Springer-Verlag, 1996.

[ATB93] N. Audsley, K. Tindell und A. Burns. The end of the line for static cyclic scheduling? In *Proceedings of the 5th Euromicro Workshop on Real-Time Systems*, Seiten 36–41. IEEE Computer Society Press, 1993.

[Aud93] N. Audsley et al. Applying new scheduling theory to static priority preemptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

[AY99] R. Alur und M. Yannakakis. Model checking of message sequence charts. In *Proc. 10th Intl. Conf. on Concurrency Theory*, Seiten 114–129. Springer Verlag, 1999.

[Bak91] T. Baker. Stack-based scheduling of real-time processes. *Real-Time Systems*, 3(1):67–99, 1991.

[BAL97] Hanene Ben-Abdallah und Stefan Leue. Expressing and analyzing timing constraints in message sequence chart specifications. Technischer Bericht, Electrical and Computer Engineering, University of Waterloo, April 1997.

[BB99] G. Bernat und A. Burns. New results on fixed priority aperiodic servers. In *Proceedings of the 20th IEEE Real-Time Symposium*, Seiten 68–78, 1999.

[BC01] Francis Bordeleau und Jean-Pierre Corriveau. On the importance of inter-scenario relationships in hierarchical state machine design. In *FASE 2001*. Springer Verlag, 2001.

[BDL+01] Gerd Behrman, Alexandre David, Kim G. Larsen, M. Oliver Möller, Paul Petterson und Wang Yi. UPPAAL - Present and Future. In *Proc. of the 40th IEEE Conference on Decision and Control*, Seiten 2281–2286, Orlando, Florida, Dezember 2001. IEEE Service Center.

[Bec01] G. Beckmann. *Ein Hochgeschwindigkeits-Kommunikations-System für die industrielle Automation.* Dissertation, Technical University of Braunschweig, Germany, November 2001.

[BFW04] Andrew J. Bennett, A. J. Field und C. Murray Woodside. Experimental evaluation of the UML profile for schedulability, performance and time. In Thomas Baar, Alfred Strohmeier, Ana Moreira und Stephen J. Mellor (Hrsg.), *UML 2004 - The Unified Modeling Language. Model Languages and Applications. 7th International Conference, Lisbon, Portugal, October 11-15, 2004, Proceedings*, Band 3273 aus *LNCS*, Seiten 143–157. Springer, 2004.

[BJNT00] Ahmed Bouajjani, Bengt Jonsson, Marcus Nilsson und Tayssir Touili. Regular model checking. In *Computer Aided Verification*, Seiten 403–418, 2000.

[BjRJ99] Grady Booch, james Rumbaugh und Ivar Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, 1999.

[BLL+95] Johan Bengtsson, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson und Wang Yi. UPPAAL - a tool suite for automatic verification of real-time systems. In *Hybrid Systems*, Seiten 232–243, 1995.

[BP99] L. Becker und C. Pereira. Simoo-rt: An integrated object-oriented environment for the development of distributed real-time systems. In *CARS & FOF'99*, 1999.

[But97] Giorgio Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications.* Kluwer Academic Publishers, 1997.

[BW01] A. Burns und A. Wellings. *Real-Time Systems and Programming Languages.* Addison-Wesley, 3rd. Auflage, 2001.

[Can99] Ercüment Canver. Model-Checking zur Analyse von Message Sequence Charts über Statecharts. UIB 99-04, Universität Ulm, 1999. ISSN 0939-5091.

[CC89] H. Chetto und M. Chetto. Some results of the earliest deadline scheduling algorithm. *IEEE Transactions on Software Engineering*, 15(10):1261–1269, 1989.

[CGHS00] K. Compton, Y. Gurevich, J. Huggins und W. Shen. An Automatic Verification Tool for UML, 2000.

[CL90] M.-I. Chen und K.-J. Lin. Dynamic priority ceilings: A concurrency control protocol for real-time systems. *Journal of Real-Time Systems*, 2(4):325–346, 1990.

[DC01] W. Damm und M. Cohen. Advanced validation techniques meet complexity challange in embedded software development, 2001.

[DFK04] Karsten Diethers, Bernd Finkemeyer und Nnamdi Kohn. Middleware zur Realisierung offener Steuerungssoftware für hochdynamische Prozesse. *it - Information Technology*, Februar 2004.

[DFKT03] Karsten Diethers, Thomas Firley, Torsten Kröger und Ulrike Thomas. A new framework for sensor based robot programming and verification. In *International Conference on Advanced Robotics*, Juni 2003.

[DGH02] Karsten Diethers, Ursula Goltz und Michaela Huhn. Model checking UML statecharts with time. In *UML 2002, Workshop on Critical Systems Development with UML*, September 2002.

[DGV02] Karsten Diethers, Ursula Goltz und Stefan Vocke. Analysis of real-time systems modeled by UML-statecharts. In *1. Int. Colloquium of the Collaborative Research Centre 562*, number 7 in Fortschritte in der Robotik. Shaker, May 2002.

[DH99] Werner Damm und David Harel. LSCs: Breathing life into Message Sequence Charts. In *3rd IFIP Int. Conference on Formal Methods for Open Object-Based Distributed Systems, (FMOODS'99)*, Seiten 293–312. Kluwer Academic Publishers, 1999.

[DHK03] Karsten Diethers, Michaela Huhn und Ivo Kasiuk. Ein Werkzeug zur automatischen Schedulability-Analyse auf Basis von UML-Modellen. In *Dresden Workshop Circuit and System Design (DASS´2003) and Workshop System Design Automation (SDA´2003)*, Mai 2003.

[Die01] Karsten Diethers. Tool-based analysis of timed sequence diagrams. Technischer Bericht 2001-01, Technical University of Braunschweig, Dezember 2001.

[DJHP97] W. Damm, B. Josko, H. Hungar und A. Pnueli. A compositional real-time semantics of statemate designs. In W.-P. de Roever, H. Langmaack und A. Pnueli (Hrsg.), *Proc. COMPOS 97*, Band 1536 aus *LNCS*. Springer-Verlag, 1997.

[DM01] Alexandre David und M. Oliver Möller. From HUPPAAL to UPPAAL: A translation from hierarchical timed automata to flat timed automata. Research Series RS-01-11, BRICS, Department of Computer Science, University of Aarhus, Marz 2001.

[DM02] Karsten Diethers und Martin Mutz. Improving software development in the automotive area through tool supported modelling and formal analysis. In *SDA 2002, System Design Automation*, Seiten 81–88, April 2002.

[DMY02] Alexandre David, M. Oliver Möller und Wang Yi. Formal verification of UML statecharts with real-time extensions. In R.-D. Kutsche und H. Weber (Hrsg.), *Fundamental Approaches to Software Engineering (FASE'2002)*, Band 2306 aus *LNCS*, Seiten 218–232. Springer-Verlag, April 2002.

[Dou98] Bruce P. Douglass. *Real-Time UML*. Addison-Wesley, 1998.

[DW95] R. Davis und A. Wellings. Dual priority scheduling. In *IEEE Proceedings of Real-Time Systems Symposium*, 1995.

[EFM97] A. Engels, L.M.G. Feijs und S. Mauw. Test generation for intelligent networks using model checking. In E. Brinksma (Hrsg.), *Proceedings of the Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, number 1217 in LNCS, Seiten 384–398. Springer-Verlag, 1997.

[EHHS00] Gregor Engels, Jan Hendrik Hausmann, Reiko Heckel und Stefan Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In Andy Evans, Stuart Kent und Bran Selic (Hrsg.), *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, Band 1939 aus *LNCS*, Seiten 323–337. Springer, 2000.

[EHK01] Gregor Engels, Reiko Heckel und Jochen Malte Küster. Rule-based specification of behavioral consistency based on the UML meta-model. In Martin Gogolla und Cris Kobryn (Hrsg.), *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools. 4th International Conference, Toronto, Canada, October 2001, Proceedings*, Band 2185 aus *LNCS*, Seiten 272–286. Springer, 2001.

[EHK03] Gregor Engels, Reiko Heckel und Jochen M. Küster. The consistency workbench: A tool for consistency management in UML-based development. In Perdita Stevens, Jon Whittle und Grady Booch (Hrsg.), *UML 2003 - The Unified Modeling Language. Model Languages and Applications. 6th International Conference, San Francisco, CA, USA, October 2003, Proceedings*, Band 2863 aus *LNCS*, Seiten 356–359. Springer, 2003.

[EHKG02] Gregor Engels, Reiko Heckel, Jochen Malte Küster und Luuk Groenewegen. Consistency-preserving model evolution through transformations. In Jean-Marc Jézéquel, Heinrich Hussmann und Stephen Cook (Hrsg.), *UML 2002 - The Unified Modeling Language. Model Engineering, Languages, Concepts, and Tools. 5th International Conference, Dresden, Germany, September/October 2002, Proceedings*, Band 2460 aus *LNCS*, Seiten 212–226. Springer, 2002.

[EHS00] Gregor Engels, Reiko Heckel und Stefan Sauer. UML – a universal modeling language? In M. Nielsen und D. Simpson (Hrsg.), *Proc. Application and Theory of Petri Nets 2000, 21st International Conference, ICATPN 2000, Aarhus, Denmark, June 2000.*, Band 1825 aus *LNCS*, Seiten 24–38. Springer, 2000.

[EHSW99] Gregor Engels, Roland Hücking, Stefan Sauer und Annika Wagner. UML collaboration diagrams and their transformation to Java. In Robert France und Bernhard Rumpe (Hrsg.), *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, Band 1723 aus *LNCS*, Seiten 473–488. Springer, 1999.

[Enc96a] Vincent Encontre. Modeling and implementing correct, scalable and efficient real-time applications with ObjectGEODE. *1rst Quarter Edition of Real-Time Magazine*, 1996.

[Enc96b] Vincent Encontre. Modeling and implementing correct, scalable and efficient real-time applications with ObjectGEODE. *1rst Quarter Edition of Real-Time Magazine*, 1996.

[EW00] Rik Eshuis und Roel Wieringa. Requirements level semantics for UML statecharts. In Scott F. Smith und Carolyn L. Talcott (Hrsg.), *Formal Methods for Open Object-Based Distributed Systems (FMOODS) 2000, IFIP TC6/WG6.1.* Kluwer Academic Publishers, 2000.

[EW01] Rik Eshuis und Roel Wieringa. A real-time execution semantics for UML activity diagrams. In Heinrich Hussmann (Hrsg.), *Fundamental Approaches to Software Engineering, 4th International Conference, FASE 2001, held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001, Genova, Italy, April 2-6, 2001, Proceedings*, Band 2029 aus *LNCS*, Seiten 76–90. Springer, 2001.

[FHD+99] Thomas Firley, Michaela Huhn, Karsten Diethers, Thomas Gehrke und Ursula Goltz. Timed sequence diagrams and tool-based analysis a case study. In Robert France und Bernhard Rumpe (Hrsg.), *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, Band 1723 aus *LNCS*, Seiten 645–660. Springer, 1999.

[Fid98] C. J. Fidge. Real-time schedulability tests for preemptive multitasking. *Journal of Real-Time Systems*, 14:61–93, 1998.

[Fin04] Bernd Finkemeyer. *Robotersteuerung auf der Basis von Aktionsprimitiven*, Band 8 aus *Fortschritte in der Informatik*. Shaker-Verlag, 2004.

[Fir04] Thomas Firley. *Computing Abstract Models for Verifying Reactive Systems*. Dissertation, TU Braunschweig, 2004.

[FJ97] Konrad Feyerabend und Bernhard Josko. A visual formalism for real-time requirement specifications. In Miquel Bertran und Teodor Rus (Hrsg.), *Transformation-Based Reactive Systems Development, Proc. 4th International AMAST Workshop on Real-Time Systems and Concurrentand Distributed Software, ARTS'97*, Band 1231, Seiten 156–168. Springer-Verlag, 1997.

[FMPY03] Elena Fersman, Leonid Mokrushin, Paul Pettersson und Wang Yi. Schedulability analysis using two clocks. In *Proceedings of 9th International Conference, TACAS'03*, Band 2619 aus *LNCS*, Seiten 224–239. Springer-Verlag, 2003.

[FPY02] Elena Fersman, Paul Pettersson und Wang Yi. Timed automata with asynchronous processes: Schedulability and decidability. In *Proceedings of 8th International Conference, TACAS 2002*, Band 2280 aus *LNCS*, Seiten 67–82. Springer-Verlag, 2002.

[Geh97] Thomas Gehrke. Zur Integration von Mehrfachkommunikation in eine prozeßorientierte Sprache mit funktionaler Datenbehandlung. Technischer Bericht, Universität Hildesheim, 1997.

[GF99] T. Gehrke und T. Firley. Generative sequence diagrams with textual annotations, 1999.

[GMP01] Elsa Gunter, Anca Muscholl und Doron Peled. Compositional message sequence charts. In *TACAS 2001*, Seiten 496–511. Springer Verlag, 2001.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[Har02] David Harel. On the behaviour of complex object-oriented systems. In Peter P. Hofmann und Andy Schurr (Hrsg.), *Conf.*

on Object-Oriented Modeling of Embedded Real-Time Systems (OMER '99), Lecture Notes in Informatics, Seiten 11–15, 2002.

[HG96] David Harel und Eran Gery. Executable object modeling with statecharts. In *Proceedings of the 18th International Conference on Software Engineering*, Seiten 246–257. IEEE Computer Society Press, 1996.

[HG97] David Harel und Eran Gery. Executable object modeling with statecharts. *IEEE Computer*, 30(7):31–42, 1997.

[HK99] D. Harel und O. Kupferman. On the behavioral inheritance of statebased objects. Technischer Bericht MCS99-12, Weizmann Institute of Science, 1999.

[HK00] David Harel und Orna Kupferman. On the behavioral inheritance of state-based objects. In *TOOLS '00: Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 34'00)*, Seite 83, Washington, DC, USA, 2000. IEEE Computer Society.

[HK04] D. Harel und H. Kugler. The Rhapsody semantics of statecharts (or, on the executable core of the uml). In H. Ehrig (Hrsg.), *Integration of Software Specification Techniques for Applications in Engineering*, Band 3147 aus *LNCS*, Seiten 325–354. Springer-Verlag, 2004.

[HLS99] Klaus Havelund, Kim Guldstrand Larsen und Arne Skou. Formal verification of a power controller using the real-time model checker UPPAAL. *Lecture Notes in Computer Science*, 1601:277–298, 1999.

[HMKT00] J.G. Henriksen, M. Mukund, K. Narayan Kumar und P.S. Thiagarajan. On Message Sequence Graphs and finitely generated regular MSC languages. In *Proceedings of the 27th International Colloquium on Automata Languages and Programming (ICALP'2000)*, number 1853 in Lecture Notes in Computer Science, Geneva, Switzerland, 2000. Springer.

[HN95] D. Harel und A. Naamad. The statemate semantics of statecharts. Technischer Bericht, Faculty of Mathematics and Computer Science, The Weizmann Institute of Science, 1995.

[HN96] David Harel und Amnon Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.

[Hol97] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[HP96] Gerard J. Holzmann und Doron Peled. The state of SPIN. In *8th International Conference on Computer Aided Verification*, Band 1102 aus *LNCS*, Seiten 385–389, New Brunswick, NJ, USA, 1996. Springer Verlag.

[HR00] David Harel und Bernhard Rumpe. Modeling languages: Syntax, semantics and all that stuff - part I: The basic stuff, 2000.

[HS01] Reiko Heckel und Stefan Sauer. Strengthening UML collaboration diagrams by state transformations. In *FASE 2001*. Springer Verlag, 2001.

[HSLL97] Klaus Havelund, Arne Skou, Kim G. Larsen und Kristian Lund. Formal modelling and analysis of an audio/video protocol: An industrial case study using UPPAAL. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, Seiten 2–13, 1997.

[Int96] International Telecommunication Union. *Message Sequence Charts (MSC)*, Z.120. Auflage, 1996.

[ITU] ITU-T. *ITU-T Recommandation Z.120* Message Sequence Chart (MSC).

[JED+94] J.R. Burch, E.M. Clarke, D.E. Long, K.L. MacMillan und D.L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, 1994.

[JEK+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill und L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In

Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science, Seiten 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[JL02] Dongxi Jin und David C Levy. An approach to schedulability analysis of uml-based real-time systems design. In *WOSP '02: Proceedings of the 3rd international workshop on Software and performance*, Seiten 243–250, New York, NY, USA, 2002. ACM Press.

[JP86] M. Joseph und P. Pandya. Finding response times in real-time systems. *BCS Computer Journal*, 29(5):390–395, 1986.

[Klo05] Karlhorst Klotz. Software ohne Fehl und Tadel. *Das M. I. T. - Magazin für Innovation Technology*, 2005.

[KMR02] Alexander Knapp, Stephan Merz und Christopher Rauh. Model checking timed UML state machines and collaborations. In W. Damm und E.-R. Olderog (Hrsg.), *7th Intl. Symp. Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT 2002)*, Band 2469 aus *Lecture Notes in Computer Science*, Seiten 395–414, Oldenburg, Germany, September 2002. Springer-Verlag.

[Kop97] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications.* Kluwer Academic Publishers, 1997.

[Krt99] Rob Krten. *Getting Started with QNX Neutrino 2.* Parse, 1999.

[KW01] Jochen Klose und Hartmut Wittke. An automata based interpretation of live sequence charts. In *TACAS 2001.* Springer Verlag, 2001.

[KY04] Pavel Krcál und Wang Yi. Decidable and undecidable problems in schedulability analysis using timed automata. In *Proceedings of 10th International Conference, TACAS'04*, Band 2988 aus *LNCS*, Seiten 236–250. Springer-Verlag, 2004.

[LE96] P. Leblanc und V. Encontre. *ObjectGeode: Method Guidelines.* VERILOG, 1996.

[Lev97] Francesca Levi. *Verification of Temporal and Real-Time Properties of Statecharts.* Dissertation, University of Pisa, 1997.

[LGT98] Agnès Lanusse, Sébastian Gérard und Francois Terrier. Real-time modelling with UML: The ACCORD approach. In *UML '98*, Band 1618 aus *LNCS*, Seiten 287–296. Springer-Verlag, 1998.

[LH99] Stefan Leue und Gerard Holzmann. v-Promela: A visual, object-oriented language for SPIN. In *Proc. of the 2nd IEEE Intern. Symp. on Object-Oriented Real-Time Distributed Computing.* IEEE Computer Society Press, 1999.

[Lil] Johan Lilius. *The Analyzer Component Framework.*

[LL73] C. Liu und J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *JACM*, 20(1):46–61, 1973.

[LL99a] Xuandong Li und Johan Lilius. Timing analysis of message sequence charts. Technischer Bericht TUCS-TR-255, Turku Centre for Computer Science, 24, 1999.

[LL99b] Xuandong Li und Johan Lilius. Timing analysis of UML sequence diagrams. In Robert France und Bernhard Rumpe (Hrsg.), *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, Band 1723, Seiten 661–674. Springer, 1999.

[LL99c] Xuandong Li und Johan Lilius. Timing analysis of UML sequence diagrams. In Robert France und Bernhard Rumpe (Hrsg.), *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, Band 1723, Seiten 661–674. Springer, 1999.

[LM87] E. A. Lee und D.G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, 36(1), 1987.

[LMM99] D. Latella, I. Majzik und M. Massink. Towards a formal operational semantics of UML statechart diagrams. In *Proc. FMOODS'99, IFIP TC6, WG6.1.* Kluwer, 1999.

[Loc92] C. Douglass Locke. Software architecture for hard real-time applications: Cyclic executives versus fixed priority executives. *The Journal of Real-Time Systems*, 4(1):37–53, 1992.

[LP] Johan Lilius und Ivan Porres. The semantics of UML state machines.

[LP99a] Johan Lilius und Ivan P Paltor. The production cell: An exercise in the formal verification of a uml model. Technischer Bericht, Turku Centre for Computer Science, 1999.

[LP99b] Johan Lilius und Ivan Porres Paltor. vUML: a tool for verifying UML models. Technischer Bericht TUCS-TR-272, Turku Centre for Computer Science, 18, 1999.

[LPP99] Johan Lilius und Ivan Porres Paltor. Formalising UML state machines for model checking. In R. France und B. Rumpe (Hrsg.), *Proc. UML'99*, Band 1723 aus *LNCS*. Springer-Verlag, 1999.

[LPW95] Kim G. Larsen, Paul Pettersson und Wang Yi. Diagnostic model-checking for real-time systems. In *Proc. of the 4th DIMACS Workshop on Verification and Control of Hybrid Systems*, Band 1066 aus *LNCS*, Seiten 575–586. Springer-Verlag, 1995.

[LPY97] Kim Guldstrand Larsen, Paul Pettersson und Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[LRD99] Kanishka Lahiri, Anand Raghunathan und Sujit Dey. Fast performance analysis of bus-based system-on-chip communication architectures. In *ICCAD*, Seiten 566–573, 1999.

[LSS87] J. Lehoczky, L. Sha und J. Strosnider. Enhanced aperiodic responsiveness in a hard real-time environment. In *Proceedings of the IEEE Real-Time Systems Symposium*, Seiten 261–270, 1987.

[LW82] J. Leung und J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 4(2):237–250, 1982.

[Lyo98] Andrew Lyons. UML for real-time overview, 1998.

[Mik] Erich Mikk. Moces user's guide.

[MLPS] E. Mikk, Y. Lakhnech, C. Petersohn und M. Siegel. On formal semantics of statecharts as supported by statemate.

[MLS97] E. Mikk, Y. Lakhnech und M. Siegel. Hierarchical automata as model for statecharts. In *Asian Computing Science Conference (ASIAN'97)*, Band 1345 aus *LNCS*. Springer Verlag, December 1997.

[MLSH98] E. Mikk, Y. Lakhnech, M. Siegel und G.J. Holzmann. Implementing statecharts in promela/SPIN. In *Proc. Workshop on Industrial-strength Formal specification Techniques*, Seiten 90–101, Boca Raton, Fl., USA, October 1998. IEEE Computer Society.

[Möl02] M. Oliver Möller. *Structure and hierarchy in real-time systems.* Dissertation, BRICS PhD school, University of Aarhus, Februar 2002.

[MR94] S. Mauw und M. A. Reniers. An algebraic semantics of basic message sequence charts. *The Computer Journal*, 37(4):269–277, 1994.

[MR97] S. Mauw und M. A. Reniers. High-level message sequence charts. In *Proceedings of the Eighth SDL Forum (SDL'97)*, Seiten 291–306, 1997.

[NS01] Brian Nielson und Arne Skou. Automated test generation from timed automata. In *TACAS 2001*. Springer Verlag, 2001.

[NS03] Marco Di Natale und Manas Saksena. *UML for real: design of embedded real-time systems*, Kapitel Schedulability analysis with UML, Seiten 241–269. Kluwer Academic Publishers, Norwell, MA, USA, 2003.

[Obj05] Object Management Group, *http://www.omg.org. UML Profile for Schedulability, Performance and Time, Version 1.1*, 2005. Version 1.1.

[OGO04] I. Ober, S. Graf und I. Ober. Model checking of uml models via a mapping to communicating extended timed automata, 2004.

[OMG00] Object Management Group, *http://www.omg.org. XML Metadata Interchange*, 2000. Version 1.1.

[OMG01] Object Management Group, *http://www.omg.org. Unified Modeling Language Specification*, 2001. Version 1.4.

[PS91] A. Pnueli und M. Shalev. What is in a step: On the semantics of statecharts. In T. Ito und A. R. Meyer (Hrsg.), *Theoretical Aspects of Computer Software*, Band 526 aus *LNCS*, Seiten 244–265. Springer-Verlag, 1991.

[RACH00] Gianna Reggio, Egidio Astesiano, Christine Choppy und Heinrich Hußmann. Analysing uml active classes and associated state machines - a lightweight formal approach. In *FASE '00: Proceedings of the Third Internationsl Conference on Fundamental Approaches to Software Engineering*, Seiten 127–146, London, UK, 2000. Springer-Verlag.

[RCA01] Gianna Reggio, Maura Cerioli und Egidio Astesiano. Towards a rigorous semantics of UML supporting its multiview approach. In *FASE 2001*. Springer Verlag, 2001.

[Ren95] M. Reniers. Static semantics of message sequence charts, 1995.

[Ren98] M. A. Reniers. *Message Sequence Chart: Syntax and Semantics*. Dissertation, Eindhoven University of Technology, Einhoven, 1998.

[SB94] M. Spuri und G. Buttazzo. Efficient aperiodic service under earliest deadline scheduling. In *Proceedings IEEE Real-Time Symposium*, Seiten 2–11, 1994.

[SBK01] Natasha Sharygina, James C. Browne und Robert P. Kurshan. A formal object-oriented analysis for software reliability: Design

for verification. In *FASE 2001*, Seiten 318–332. Springer Verlag, 2001.

[SE99] Stefan Sauer und Gregor Engels. UML-basierte Modellierung von Multimediaanwendungen. In J. Desel, K. Pohl und A. Schürr (Hrsg.), *Proc. Modellierung'99, March 10-12, 1999, Karlsruhe, Germany*, Seiten 155–170. Teubner, Stuttgart, 1999.

[Sel98] Bran Selic. Using UML for modeling complex real-time systems. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, Seiten 250–260. Springer-Verlag, 1998.

[SGME92] Bran Selic, Garth Gullekson, Jim McGee und Ian Engelberg. Room: An object-oriented methodology for developing real-time systems. In *CASE'92, Fifth Intern.Workshop on Computer-Aided Software Engineering*, 1992.

[SGW94a] Bran Selic, Garth Gullekson und Paul T. Ward. *Real-Time Object-Oriented Modeling.* John Wiley & Sons, 1994.

[SGW94b] Bran Selic, Garth Gullekson und Paul T. Ward. *Real-time object-oriented modeling.* John Wiley & Sons, Inc., 1994.

[SP01] Natasha Sharygina und Doron Peled. A combined testing and verification approach for software reliability. In *FME*, Seiten 611–628. Springer Verlag, 2001.

[SR88] John A. Stankovic und Krithi Ramamritham. Tutorial: Hard real-time systems, 1988.

[SRL90] L. Sha, R. Rajkumar und J. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[SRM97] Ina Schieferdecker, Axel Rennoch und Olaf Merkens. Timed MSCs - an extension to MSC'96. In A. Wolisz, I. Schieferdecker und A. Rennoch (Hrsg.), *Formale Beschreibungstechniken für Verteilte Systeme, GMD-Studie Nr. 315*, 1997.

[SRS94] L. Sha, R. Rajkumar und S. Sathaye. Generalized rate-monotonic scheduling theory: A framework for developing real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, Seiten 68–82, 1994.

[SSL89] B. Sprunt, L. Sha und J. Lehoczky. Aperiodic task scheduling for hard real-time systems. *The Journal of Real-Time Systems*, 1:27–60, 1989.

[Sta98] J. A. Stankovic et al. *Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms*. Kluwer Academic, 1998.

[SvG98] J. Seemann und J. Wolff von Gudenberg. Extension of UML Sequence Diagrams for real-time systems. In *UML '98*, Band 1618 aus *LNCS*, Seiten 225–233. Springer-Verlag, 1998.

[TBW94] K. Tindell, A. Burns und A. Wellings. An extendible approach for analysing fixed priority hard real-time tasks. *Real-Time Systems*, 6(2):133–151, 1994.

[TC94] K. Tindell und J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming – Euromicro Journal*, 40:117–134, 1994.

[Tin94] K. Tindell. Adding time-offsets to schedulability analysis. Technischer Bericht, Department of Computer Science, University of York, England, 1994.

[vdB94] M. von der Beeck. A comparison of statecharts variants. In H. Langmaack, W.-P. de Roever und J. Vytopil (Hrsg.), *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Band 863 aus *LNCS*, Seiten 128–148. Springer-Verlag, 1994.

[VGP01] Dániel Varró, Szilvia Gyapay und András Pataricza. Automatic transformation of UML models for system verification. In Jon Whittle et al. (Hrsg.), *WTUML'01: Workshop on Transformations in UML*, Seiten 123–127, Genova, Italy, April 7th 2001.

[Wan04] F. Wang. Formal verification of timed systems: A survey and perspective. *Proc. of the IEEE*, 92(8), August 2004.

[WB98] Roel Wieringa und Jan Broersen. A minimal transition system semantics for lightweight class- and behavior diagrams. In Manfred Broy, Derek Coleman, Tom S. E. Maibaum und Bernhard Rumpe (Hrsg.), *Proceedings PSMT'98 Workshop on Precise Semantics for Modeling Techniques*. Technische Universität München, TUM-I9803, 1998.

[WT04] Shuhua Wang und Grace Tsai. Specification and timing analysis of real-time systems. *Real-Time Systems*, 28(1):69–90, Oktober 2004.

[Yov97] Sergio Yovinc. *Kronos: A Verification Tool for Real-Time Systems*, 1997.

[ZA95] Z. Manna und A. Pnueli. Clocked Transition Systems. In *Workshop on Verification and Control of Hybrid*, New Brunswick, NJ., 1995.

Index